James Noble (Ed.)

ECOOP 2012 – Object-Oriented Programming

LNCS 7313

26th European Conference
Beijing, China, June 2012
Proceedings

Lecture Notes in Computer Science 7313

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

James Noble (Ed.)

ECOOP 2012 – Object-Oriented Programming

26th European Conference
Beijing, China, June 11-16, 2012
Proceedings

Volume Editor

James Noble
Victoria University of Wellington
School of Engineering and Computer Science
Wellington 6140, New Zealand
E-mail: kjx@ecs.vuw.ac.nz

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31056-0 e-ISBN 978-3-642-31057-7
DOI 10.1007/978-3-642-31057-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012939227

CR Subject Classification (1998): D.1.5, D.1-3, F.3, C.2, F.4, J.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

If objects did not exist, we would have to invent them.

In the last few days, I've been reading a couple of interesting pieces, ostensibly on programming without objects. The first of these is Doug Hoyte's *Let over Lambda*—"one of the most hardcore computer programming books out there"—according to the back-cover copy, and certainly an interesting and engaging read. In six chapters and 200 pages, we start in Lisp and move from closures to `lambda` expressions to `alambdas`, `dlambdas` and ultimately to `plambdas`. These "Pandoric Lambdas" create a closure that can respond to different methods, and whose state can either be encapsulated or visible outside. I seem to remember SIMULA had something similar in the mid-1960s.

The second is Jonathan Shapiro's *Retrospective Thoughts on BitC* (posted to `bitc-dev` on March 23, 2012). Of the four reasons why Shapiro chose to abandon BitC, the third is "The absence of some form of inheritance," and Shapiro goes on to say "I could nearly imagine getting what I needed by adding `ThisType` and inherited interfaces. But ... the combination is equivalent (from a type system perspective) to single-inheritance subclassing."

Truly, if objects did not exist, we would have to invent them. At ECOOP, of course, we have an advantage: Ole-Johan Dahl and Kristen Nygaard did invent objects; Alan Kay built a dynamic language based on objects; and the rest is the history of a large fraction of practical and academic programming for the last 30 years.

This year's ECOOP continued in that tradition, and started some new traditions of its own. ECOOP 2012 was only the second ECOOP to be held outside Europe (OOPSLA/ECOOP 1990 was held in Ottawa, and Canada is not technically part of Europe); ECOOP 2012 was the first ECOOP to be held in Asia; the first to be co-located with another programming language conference (PLDI); and the first to have a Program Chair from New Zealand. I must admit I was not entirely sure how that combination of circumstances would affect the conference. As far as the technical program represented in this volume is considered, this has been a great success: 140 papers were submitted, a significant increase over the last few ECOOPs.

Each paper was allocated to at least three Program Committee members to review — some papers were allocated more. All in all, we received 466 reviews, including external reviews contributed by 104 external reviewers. The Program Committee discussed these reviews online, after which authors had the opportunity to respond to reviews. The Program Committee then met in London and selected the 30 papers presented here. Of the 140 submissions, 16 were (co-)authored by members of the Program Committee. These papers received at least five reviews, and four of them were accepted.

A conference is only as good as the research it presents. I would like to thank all the authors who submitted their work to ECOOP: without your courage in sending your work, there would be no conference! I would like to thank the Program Committee, who collectively read and evaluated every paper submitted, and provided as much feedback as they could manage to the papers' authors. The quality of the Program Committee has long been a strength of ECOOP, and this year was no exception. Chairing the committee has been an honor and a privilege.

Thanks are due to Tony Hosking and Hong Mei, ECOOP Conference Chairs, for actually organizing the conference; to Tony Hosking (again), to Sophia Drossopoulou and Susan Eisenbach for organizing and hosting the PC meeting; and to Richard van de Stadt for CyberChairPRO. Tony (again) and Steve Blackburn chaired discussions on papers where I had a conflict of interest. Finally, thanks are due to Jan Vitek, Co-chair of PLDI 2012, who first suggested collocating ECOOP and PLDI in Beijing. That seems like a great decision (so far).

And now, all that remains is to ignore the talks, read email through the keynotes, sleep through the summer school, tweet through the tutorials, disregard the workshops, and enjoy all the many and varied sights and delights of Beijing, sure in the knowledge that when we return home, these proceedings will still be waiting for us — the twenty-sixth of their kind, this year's modest addition to the history of object-oriented programming.

March 2012 James Noble

Organization

ECOOP 2012 was organized by the Computer Science Department of Purdue University, under the auspices of AITO (Association Internationale pour les Technologies Objets), and in cooperation with ACM SIGPLAN and ACM SIGSOFT.

Conference Co-chairs

Antony Hosking	Purdue University, USA
Hong Mei	Peking University, China

Program Chair

James Noble	Victoria University of Wellington, New Zealand

Local Organizing Co-chairs

Lu Zhang	Peking University, China
Hongyu Zhang	Tsinghua University, China

Publicity Chair

Tao Xie	North Carolina State University, USA

Workshop Chair

Adam Welc	Adobe, USA

Workshop Co-chair

Patrick Eugster	Purdue University, USA

Summer School Chair

Jan Vitek	Purdue University, USA

Student Volunteer Co-Chairs

Max Schaefer	University of Oxford, UK, and IBM Research, USA
Xiaoying Bai	Tsinghua University, China

Web Chair

Ahmed Hussein	Purdue University, USA

Silver Sponsors

Bronze Sponsors

Program Committee

Elisa Baniassad	Australian National University, Australia
Gavin Bierman	Microsoft Research, UK
Steve Blackburn	Australian National University, Australia
John Tang Boyland	University of Wisconsin-Milwaukee, USA
Nick Cameron	Victoria University of Wellington, New Zealand
Shigeru Chiba	Tokyo Institute of Technology, Japan

Siobhán Clarke	Trinity College Dublin, Ireland
Yvonne Coady	University of Victoria, Canada
Wolfgang De Meuter	Vrije Universiteit Brussel, Belgium
Matthew B. Dwyer	University of Nebraska - Lincoln, USA
Matthew Flatt	University of Utah, USA
Neal Glew	Intel, USA
Kathryn E. Gray	University of Cambridge, UK
Matthias Hauswirth	University of Lugano, Switzerland
Robert Hirschfeld	Hasso-Plattner-Institut Potsdam, Germany
Atsushi Igarashi	Kyoto University, Japan
Bart Jacobs	Katholieke Universiteit Leuven, Germany
Richard Jones	University of Kent, UK
K. Rustan M. Leino	Microsoft Research, USA
Nick Mitchell	IBM Research, USA
Robert O'Callahan	Mozilla Corporation, New Zealand
Jens Palsberg	University of California, Los Angeles, USA
John Potter	The University of New South Wales, Australia
Ganesan Ramalingam	Microsoft Research, India
Dirk Riehle	Friedrich-Alexander University of Erlangen-Nurnberg, Germany
Yannis Smaragdakis	University of Massachusetts, Amherst, USA, and University of Athens, Greece
Eelco Visser	Delft University of Technology, The Netherlands
Tobias Wrigstad	Uppsala University, Sweden
Hongseok Yang	University of Oxford, UK
Jianjun Zhao	Shanghai Jiao Tong University, China
Elena Zucca	University of Genova, Italy

External Reviewers

Amal Ahmed	Walter Cazzola
John Altidor	Maura Cerioli
Davide Ancona	Tom Van Cutsem
Malte Appeltauer	Theo D'Hondt
Beatrice Åkerblom	Danny Dig
George Balatsouras	Tom Dinkelaker
Nick Benton	Hannes Dohrn
Carl Friedrich Bolz	Stefan Engblom
Silvia Bonomi	Anthony Estey
Johannes Borgström	Shayne Flint
Sebastian Burckhardt	Celina Gibbs
Nicolas Cardozo	Matt Giles
Andoni Lombide Carreton	Aaron Greenhouse
Federico Cavalieri	Danny M. Groenewegen

Giovanna Guerrini
Sebastian Günther
John Hawthorn
Chris Hayden
Dave Herman
Michael Hicks
Lode Hoste
Cubtaro Igarashi
Lintaro Ina
Alan Jeffrey
Adrian Johnstone
Niels Joncheere
Peter A. Jonsson
Tetsuo Kamina
George Kastrinis
Lennart Kats
Rob Kelly
Andrew Kennedy
Liam Kiemele
Carsten Kolassa
Neelakantan Krishnaswami
Giovanni Lagorio
Doug Lea
I-Ting Angelina Lee
Jens Lincke
Donna Long
Geoffrey Mainland
Dmitri Makarov
Chris Matthews
Christopher McKnight
Mojtaba Mehrara
Francesco Zappa Nardelli
Jens Nicolay
Dominic Orchard
Scott Owens
Johan Östlund
Matthew Parkinson
Javier Perez
Michael Perscheid
Tomas Petricek
Dean Pucsek
William Retert
Noam Rinetzky
Claudio Russo
Chieri Saito
Michel A. Salim
Adrian Schroeter
Jaroslav Sevcik
Jeremy Siek
Daniel Spiewak
Manu Sridharan
Bastian Steinert
Reinout Stevens
Kohei Suenaga
Alexander J. Summers
Chao Sun
Qiang Sun
Marcel Taeumel
Ryan Tandy
Martin Tillenius
Sam Tobin-Hochstadt
Laurence Tratt
Stephen Tredger
Viktor Vafeiadis
Jorge Vallejos
Dimitris Vardoulakis
Mattias De Wael
Dennis Wagelaar
Luke Wagner
Jeff Walden
Brian Warner
Feng Xie
Xi Yang
Greta Yorsh
Cheng Zhang
Sai Zhang

Table of Contents

Keynote 1

Extensibility

Language Evaluation

Ownerhsip and Initialisation

Keynote 2: Dahl-Nygaard Junior Award Winner

Language Features

Special-Purpose Analyses

JavaScript

Hardcore Theory

Modularity

Updates and Interference

General-Purpose Analyses

When Compilers Are Mirrors

Martin Odersky

EPFL
martin.odersky@epfl.ch
http://lampwww.epfl.ch/~odersky

Abstract. When compilers are reflective mirrors, interesting things happen. Reflection and compilers do tantalizing similar things. Yet, in mainstream, statically typed languages the two have been only loosely coupled, and generally share very little code. In this talk I explore what happens if one sets out to overcome their separation.

The first half of the talk addresses the challenge how reflection libraries can share core data structures and algorithms with the language's compiler without having compiler internals leaking into the standard library API. It turns out that a component system based on abstract types and path-dependent types is a good tool to solve this challenge. I'll explain how the "multiple cake pattern" can be fruitfully applied to expose the right kind of information.

The second half of the talk explores what one can do when strong, mirror-based reflection is a standard tool. In particular, the compiler itself can use reflection, leading to a particular system of low-level macros that rewrite syntax trees. One core property of these macros is that they can express staging, by rewriting a tree at one stage to code that produces the same tree at the next stage. Staging lets us implement type and abstract syntax tree reification. What's more, staging can also be applied to the macro system itself, with the consequence that a simple low-level macro system can produce a high-level hygienic one, without any extra effort from the language or compiler.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, p. 1, 2012.

Extensibility for the Masses
Practical Extensibility with Object Algebras

Bruno C.d.S. Oliveira[1] and William R. Cook[2]

[1] National University of Singapore
bruno@ropas.snu.ac.kr
[2] University of Texas, Austin
wcook@cs.utexas.edu

Abstract. This paper presents a new solution to the expression problem (EP) that works in OO languages with simple generics (including Java or C#). A key novelty of this solution is that advanced typing features, including F-bounded quantification, wildcards and variance annotations, are not needed. The solution is based on *object algebras*, which are an abstraction closely related to algebraic datatypes and Church encodings. Object algebras also have much in common with the traditional forms of the Visitor pattern, but without many of its drawbacks: they are extensible, remove the need for accept methods, and do not compromise encapsulation. We show applications of object algebras that go beyond toy examples usually presented in solutions for the expression problem. In the paper we develop an increasingly more complex set of features for a mini-imperative language, and we discuss a real-world application of object algebras in an implementation of remote batches. We believe that object algebras bring extensibility to the masses: object algebras work in mainstream OO languages, and they significantly reduce the conceptual overhead by using only features that are used by everyday programmers.

1 Introduction

The "expression problem" (EP) [38,10,46] is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants. Traditionally the kinds of data abstraction found in functional languages can be extended with new operations, but adding new data variants is difficult. The traditional object-oriented approach to data abstraction facilitates adding new data variants (classes), while adding new operations is more difficult. The Visitor Pattern [13] is often used to allow operations to be added to object-oriented data abstractions, but the common approach to visitors prevents adding new classes. Extensible visitors can be created [43,50,31], but so far solutions in the literature require complex and unwieldy types, or advanced programming languages.

In this paper we present a new approach to the EP based on *object algebras*. An object algebra is a class that implements a generic abstract factory interface, which corresponds to a particular kind of *algebraic signature* [18]. Object

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 2–27, 2012.

algebras are closely related to the Abstract Factory, Builder and Visitor patterns and can offer improvements on those patterns. Object algebras have strong theoretical foundations, inspired by earlier work on the relation between Church encodings and the Visitor pattern [5,30,35,31].

Object algebras use simple, intuitive generic types that work in languages such as Java or C#. They do not need the most advanced and difficult features of generics available in those languages, e.g. F-bounded quantification [6], wildcards [44] or variance annotations. As a result, object algebras are applicable to a wide range of programming languages that have basic support for generics.

An important advantage of object algebras over traditional visitors is that there is no need for `accept` methods. As a consequence object algebras support *retroactive implementations* [47] of interfaces or operations without preparation of existing source code. This is unlike the Visitor pattern, which can only provide retroactive implementations if the original classes include `accept` methods.

We discuss applications of object algebras that go beyond toy examples usually presented in solutions for the EP. In the paper an increasingly more complex set of features for a mini-imperative language and a real-world application of object algebras in an implementation of remote batches [22,48] are described.

Object algebras have benefits beyond the basic extensibility of the EP. They can address harder related problems, including the *expression families problem* (EFP) [31], *family polymorphism* [12] and *independent extensibility* [50].

Programming with object algebras does require learning new design strategies. Rather than creating generic objects and then visiting them to perform operations, object algebras encourage that object creation is done relative to a factory, so that specialized factories can be defined to create objects with the required operations in them. Programming against factories has some cost to it because it requires parametrization of code by factories and uses of generic types. However there are significant benefits in terms of flexibility and extensibility and, in comparison with other solutions to the EP using generic types [46,3,43,50,31], the additional cost is significantly smaller.

In summary, our contributions are:

- A solution to the EP using simple generic types. The solution can be used in mainstream languages such as Java or C#. We use Java in this paper.
- An alternative to the Visitor pattern that avoids many of the disadvantages of that pattern: it eliminates the need for `accept` methods; does not require preparation of the "visited" classes; and it supports extensibility.
- Various techniques for dealing with challenges that arise in realistic applications. For example, multi-sorted object algebras deal with multiple recursive types and generic combinator classes deal with independent extensibility.
- Insights on the relation between the Abstract Factory and Visitor patterns. In some sense, factories and visitors are two faces of object algebras.
- Case study using remote batches. The Java implementation is available online at `batches.wikidot.com`. Code for the smaller Java examples, as well as solutions to the expression problem in other languages, is available at `http://ropas.snu.ac.kr/~bruno/oa`.

```
interface Exp {
  Value eval();
}
class Lit implements Exp {
  int x;
  public Lit(int x) { this.x = x; }

  public Value eval() {
    return new VInt(x);
}}
class Add implements Exp {
  Exp l, r;
  public Add(Exp l, Exp r) { this.l = l; this.r = r; }

  public Value eval() {
    return new VInt(l.eval().getInt() + r.eval().getInt());
}}
```

Fig. 1. An object-oriented encoding of integer expressions

2 Background

While there is extensive literature on the expression problem and abstract algebra in programming languages, we summarize the required background here.

2.1 The Expression Problem

Wadler's [46] formulation of the expression problem prescribes four requirements for potential solutions. Zenger and Odersky [50] add an extra requirement (independent extensibility) to that list. These requirements are summarized here:

- *Extensibility in both dimensions*: A solution must allow the addition of new data variants and new operations and support extending existing operations.
- *Strong static type safety*: A solution must prevent applying an operation to a data variant which it cannot handle using static checks.
- *No modification or duplication*: Existing code must not be modified nor duplicated.
- *Separate compilation and type-checking*: Safety checks or compilation steps must not be deferred until link or runtime.
- *Independent extensibility*: It should be possible to combine independently developed extensions so that they can be used jointly.

To illustrate the difficulty of solving the expression problem, we review two standard forms of extensibility in object-oriented languages and show how they fail to solve the problem.

Figure 1 shows an attempt to solve the EP using polymorphism. The basic idea is to define an interface `Exp` for expressions with an evaluation operation in it, and then define concrete implementations (data variants) of that interface

for particular types of expressions. Note that evaluation returns a value of type `Value`. For the purposes of this paper we assume the following definitions of `Value` and its subclasses:

```
interface Value {
  Integer getInt();
  Boolean getBool();
}
class VInt implements Value {...}
class VBool implements Value {...}
```

It is easy to add new data variants to the code in Figure 1, but adding new operations is hard. For example, supporting pretty printing requires modifying the `Exp` interface and its implementations to add a new method. However this violates "no modification" requirement. While inheritance can be used to add new operations, the changes must be made to the interface and all classes simultaneously, to ensure static type safety. Doing so is possible, but requires advanced typing features.

An alternative attempt uses the VISITOR pattern [13]. The VISITOR pattern makes adding new operations easy, although a different interface for expressions is required:

```
interface Exp {
  <A> A accept(IntAlg<A> vis);
}
```

The `IntAlg` visitor interface, defined in Figure 2, has a (visit) method for each concrete implementation of `Exp`. These visit methods are used in the definitions of the `accept` methods. For example, the definition of the `Add` class would be:

```
class Add implements Exp {
  Exp left, right;
  public Add(Exp left, Exp right) { this.left = left; this.right = right; }

  public <A> A accept(IntAlg<A> vis) {
    return vis.add(left.accept(vis), right.accept(vis));
}}
```

There are several kinds of visitors in the literature [35]. We use a (functional) *internal visitor* [5,35] for our example since this type of visitors will be important later in Section 5.1. An internal visitor is a visitor that produces a value by processing the nodes of a composite structure, where the control flow is controlled by the infrastructure rather than the visitor itself.

With visitors, new operations are defined in concrete implementations of visitor interfaces like `IntAlg`. Figure 5 shows a concrete visitor for pretty printing. Unlike the first solution, adding new operations can be done without modifying `Exp` and its implementations. This is especially important when dealing with objects created by library classes, because it is often impossible to change the code of the library. From a software engineering viewpoint, Visitors localize code for operations in one place, while conventional OO designs scatter code for operations across multiple classes. Visitors also provide a nice way to have state that is local to an operation (rather than to a class).

```
interface IntAlg<A> {
  A lit(int x);
  A add(A e1, A e2);
}
```

Fig. 2. Visitor interface for arithmetic expressions (also an object algebra interface)

Unfortunately, traditional visitors trade one type of extensibility for another: adding new data variants is hard with visitors. The problem is the concrete references to visitor interfaces `IntAlg` in the `accept` method. Adding new data variants requires modifying `IntAlg` and all its implementations with new visit methods to deal with the new variants. Another drawback of visitors is that some initial preparation is required: the visited classes need to provide an `accept` method. This can be a problem when the source code of the classes that we want to visit is not available: if the classes have no `accept` method it is impossible to use the VISITOR pattern.

2.2 Algebraic Signatures, F-Algebras, and Church Encodings

An *algebraic signature* Σ [18] defines the names and types of functions that operate over one or more abstract types, called *sorts*. We assume the existence of some primitive built-in sorts for integers and booleans.

signature E
  lit: Int $\rightarrow$ E
  add: E $\times$ E $\rightarrow$ E

A general algebraic signature can contain *constructors* that return values of the abstract set, as well as *observations* that return other kinds of values. In this paper we restrict signatures to only contain constructors, as in the example given above. We call such signatures *constructive*.

An Σ-*algebra* is a set together with a collection of functions whose type is specified in the signature Σ. A given signature can have many algebras. For example, one valid E-algebra has a set of two values and simple constant operations: (E=$\{x, y\}$, lit=$\lambda n.x$, add=$\lambda(a, b).x$), where x, y are arbitrary constants. This algebra seems unsatisfying because it is degenerate, in that it ignores the inputs of its functions, and messy, in that its set includes extra values that are never used. A special algebra, called the *initial* or *free* algebra, is neither messy nor degenerate. One way to create the initial algebra is to use a set that contains expressions, which are applications of functions in all legal ways according to the signature, and to define the functions simply as constructors. The initial algebra looks like this:

E = { `lit`(0), `lit`(1), ..., `add`(`lit`(0), `lit`(0)), `add`(`lit`(0), `lit`(1)), ... }
lit = λn.`lit`(n)
add = $\lambda(a, b)$.`add`(a, b)

The concept of a constructive signature defined above is a syntactic characterization of a class of algebras. A more fundamental approach comes from merging

the signature's constructor functions $f_1 : T_1 \to A, ..., f_n : T_n \to A$ into a single function $f : F(A) \to A$ where F is a functor given by $F(A) = T_1 + ... + T_n$. This transformation is based on the isomorphism $(S+T) \to A \approx (S \to A) \times (T \to A)$. The function $f : F(A) \to A$ is called an F-algebra. When F is a functor built of sums and products, it can be used to give a (categorical) semantics to algebraic datatypes [26]. For example, the functor for integer expressions is $F(E) = \text{Int} + (E \times E)$. The free algebra is then the initial algebra in the category of F-algebras. Because F-Algebras provide a nice framework to formalize and reason about algebraic datatypes, they have been widely explored by the functional programming community.

It is also possible to define free algebras in a complete different way, by using Church encodings. Church encodings involve converting the algebra signature into a particular kind of polymorphic type [17]. For example, given a signature Σ with with sort A and functions $f_1 : T_1 \to A, ..., f_n : T_n \to A$, the Church encoding is given by the type

$\text{Church}_\Sigma = \forall A.(T_1 \to A) \times ... \times (T_n \to A) \to A$

A Church encoding works by taking an algebra (sort and functions) as input and using it to create an element of the sort. Thus a Church "value" is not really a value, but rather a *recipe* for creating a value. The recipes in a Church encoding are isomorphic to the free algebra because of parametericity [15]. As a concrete example, the signature E defined above has the Church encoding:

$\text{Church}_E = \forall E.(Int \to E) \times (E \times E \to E) \to E$

When interpreted in object-oriented programming, Church encodings correspond to internal visitors [5,35]. From a functional programming point of view, Church encodings represent data as folds [15].

3 Object Algebras

Algebraic signatures can be defined in statically typed object-oriented languages by creating a generic interface whose parameter is the abstract type. We call an interface representing an algebraic signature an *object algebra interface*. An example of an object algebra interface representing the abstract syntax of simple expressions is given in Figure 2, which was previously introduced as the type of an internal visitor. Object algebra interfaces correspond closely to Abstract Factory interfaces [13]. The difference is that a factory interface typically uses a specific concrete class or interface as the result type for the factory methods, while the object algebra interface has a generic type. The factory interface can be derived by instantiating the abstract type to the specific object interface of the objects being created.

An *object algebra* is a class that implements an object algebra interface. Figure 3 defines an object algebra that plays the role of a factory for expressions. The factory defines how to create each kind of object in the composite structure.

To create an actual object, some part of the code will instantiate the factory and then invoke its methods repeatedly to create a specific instance. This object construction process may also be *parameterized* by the factory itself, allowing the

```
class IntFactory implements IntAlg<Exp> {
  public Exp lit(int x) {
    return new Lit(x);
  }
  public Exp add(Exp e1, Exp e2) {
    return new Add(e1, e2);
}}
```

Fig. 3. Using an object algebra as a factory

process to create specific objects using different factories. The result is similar to a Church encoded value. For example, a function to create an expression object, and an example test function that uses it, are given below.

```
<A> A make3Plus5(IntAlg<A> f) {
  return f.add( f.lit(3), f.lit(5) );
}
void test() {
  Exp e = make3Plus5(new IntFactory());
}
```

Note that a similar function could be written to parse expressions or load them from a binary representation. For example, the following function parses an integer expression from a string.

```
<A> A parseExp(IntAlg<A> f, String s) {
   if (s.equals("0"))
     return f.lit(0);
   else {... /* more interesting parsing cases */}
}
```

4 Retroactive Interface Implementations

This section shows one of the key advantages of object algebras: support for retroactive interface implementations without requiring initial preparation of code.

To illustrate retroactive implementations consider the simple object-oriented implementation of arithmetic expressions in Figure 1. These expressions support evaluation, but not pretty printing. Suppose that we now wanted to support pretty printing. Normally, as discussed in Section 2.1, we would either:

1. change the definition of the interface `Exp` to support a printing operation and change all the implementors of that interface to implement the operation; or
2. use the VISITOR pattern, which would also require modifications in the class hierarchy to introduce `accept` methods.

Both options require pervasive changes to existing code. Furthermore, these changes are only an option if the source code is available. If the hierarchy is part of a library or framework, then these solutions are not options.

```
interface IPrint {
  String print();
}
class IntPrint implements IntAlg<IPrint> {
  public IPrint lit(final int x) {
    return new IPrint() {
      public String print() {
        return new Integer(x).toString();
    }};
  }
  public IPrint add(final IPrint e1, final IPrint e2) {
    return new IPrint() {
      public String print() {
        return e1.print() + " + " + e2.print();
    }};
}}
```

Fig. 4. A retroactive implementation of printing for arithmetic expressions

What we would like is a mechanism that allowed us to retroactively implement interfaces for existing class hierarchies, without need for changes to existing implementations.

As it turns out object algebras enable us to simulate such retroactive implementations of interfaces. To use object algebras to provide retroactive implementations we proceed very much like an implementation of internal visitors. The idea is illustrated in Figure 4. To provide the retroactive implementation of the interface, we create an implementation of the object algebra with the abstract type instantiated to the interface type. In this case `IPrint` is the interface that the arithmetic expressions should implement. The interface implementations are done by creating a class `IntPrint` that implements the object algebra interface `IntAlg<IPrint>`. The implementations of the two methods `lit` and `add` provide the implementation of the interface for literals and addition.

The difference to the VISITOR pattern is that we do not add `accept` methods to `Exp`. Instead, following the approach presented in Section 3, we replace uses of concrete constructors in the client code by the corresponding methods in the object algebra. For example, instead of creating an expression

```
Exp exp = new Add(new Lit(3), new Lit(4));
```

we would abstract uses of the constructors as follows:

```
<A> A exp(IntAlg<A> v) {
  return v.add(v.lit(3), v.lit(4));
}
```

With this transformation in place we could then write the following code:

```
void test() {
  IntFactory base = new IntFactory();
  IntPrint print = new IntPrint();

  int x = exp(base).eval(); // int x = exp.eval();
```

```
  String s = exp(print).print();
}
```

Compared to the conventional object-oriented style, uses of `exp.m()` are replaced by `exp(mFactory).m()`, where `mFactory` is a factory that creates objects with the required `m` method.

By using this simple pattern we can provide retroactive implementations of interfaces to existing code. In comparison to Java extensions such as JavaGI [47], which provide native language support for retroactive implementations, there is of course some overhead in terms of additional code. On the other hand, no new compiler is needed. One difficulty of using this pattern arises when the operations in the retroactive interface implementations depend on existing operations in the base classes or other retroactive implementations. The simple pattern presented in this section is insufficient to allow such dependencies. However, with a bit more work, we can get around this restriction as we shall see in Section 7.3.

Finally, note that this style of retroactive implementations is quite powerful: it still allows us to simulate *dynamically* dispatched methods and open classes [9]. This is unlike approaches such as C# extension methods [1] or conventional object-oriented encodings of type classes [34] which can only provide *static* dispatching in their retroactive implementations. Although the programming style required by object algebras (and retroactive interface implementations) is similar to the use of object-oriented encodings of type classes, the key difference is that object algebras overload *constructors* instead of regular methods.

5 Extensibility

There are two ways in which we may want to extend our expressions: adding new variants; or adding new operations. The previous section has already shown one way in which we can add new operations: via retroactive interface implementations. In this section we show another alternative way to define operations and illustrate the addition of new variants. We also show how object algebras go beyond many solutions to the EP and also provide solution to the *expression families problem* [31].

5.1 Internal Visitors as Object Algebras

Object algebras provide a direct implementation of (functional) internal visitors [35] (see also Section 2.1) since constructive algebraic signatures correspond exactly to internal visitor interfaces. As such we can use object algebras to define new operations using concrete internal visitor implementations. As Figure 5 shows this offers an alternative way to implement pretty printing. Instead of creating a new interface like `IPrint` and defining a retroactive implementation for that type, we can directly define the printing operation. In this case it is the `IntAlg` interface that is interpreted as an internal visitor interface.

```
class Print2 implements IntAlg<String> {
  public String lit(int x) {
    return new Integer(x).toString();
  }

  public String add(String e1, String e2) {
    return e1 + " + " + e2;
  }
}
```

Fig. 5. Adding a printing operation

```
interface IntBoolAlg<A> extends IntAlg<A> {
  A bool(Boolean b);
  A iff(A e1, A e2, A e3);
}
```

Fig. 6. Adding boolean expression variants

Printing is used as follows:

```
Print2 p = new Print2();
String s = exp(p);
```

This object algebra visitor style avoids the creation of an intermediate object, just to immediately invoke the `print` method afterwards. Unlike traditional visitor implementations, this visitor style using object algebras supports data variant extensibility and does not need `accept` methods.

Using internal visitors is best when the computation in the operation happens bottom-up: essentially operations that could be defined as folds in functional programming. This stems, of course, from the fact that internal visitors are basically Church encodings and Church encodings encode data as folds. For operations that do not naturally fit this bottom-up style of computation, or mutually depend on other operations, the factory-oriented approach using retroactive implementations of interfaces is better.

5.2 Adding New Variants and Updating Operations

Adding new data variants is easy. The first step is to create new classes `Bool` and `Iff` in the usual object-oriented style (like `Lit` and `Add`):

```
class Bool implements Exp {...}
class Iff implements Exp {...}
```

The second step, shown in Figure 6, is to create an extended algebra interface with two new methods for the new boolean expressions. Finally the last step, shown in Figure 7, is to provide extension for the new boolean expressions cases for both the factory `IntFactory` and the retroactive implementation for printing.

```
/* Extended Expression Factory */
class IntBoolFactory extends IntFactory implements IntBoolAlg<Exp> {
  public Exp bool(Boolean b) {return new Bool(b);}

  public Exp iff(Exp e1, Exp e2, Exp e3) {return new Iff(e1,e2,e3);}
}

/* Extended Retroactive Implementation for Printing */
class IntBoolPrint extends IntPrint implements IntBoolAlg<IPrint> {
  public IPrint bool(final Boolean b) {
    return new IPrint() {
      public String print() {return new Boolean(b).toString();}
    };
  }

  public IPrint iff(final IPrint e1, final IPrint e2, final IPrint e3) {
    return  new IPrint() {
      public String print() {
        return "if (" + e1.print() + ") then " + e2.print() + " else " + e3.
            print();
      }
    };
  }
}
```

Fig. 7. Supporting boolean expression variants

5.3 Subtyping Relations

There are two interesting subtyping relations when we talk about the EP: subtyping between extended and base terms; and subtyping between the operations on those terms. Object algebras support both types of subtyping.

Not many other solutions to the EP support such subtyping relations. Even using advanced features like virtual classes [25] and similar mechanisms [4,27,29], the extended terms and operations are incompatible with the base terms and operations: only subtyping relations between classes in the *same family* are preserved. Oliveira [31] recognized this problem and suggested a variant of the EP: the *expression families problem*, which requires solutions to preserve the subtyping relations *across different families*. There are two solutions that we are aware of that do support such subtyping relations [43,31]. Still both of them require variance annotations or wildcards.

Subtyping between object algebra interfaces follows from standard OO subtyping: an extension of an object algebra interface is a *subtype* of the original interface. A consequence of this subtyping relation is that if we have some term constructed using a certain object algebra, we can always use an operation defined over an extension of that object algebra to process the term. For example:

```
IntBoolPrint p2 = new IntBoolPrint();
exp(p2).print();
```

In this case we can use `p2` (which supports integer and boolean expressions) in an integer expression. However, the following code would be rejected:

```
IntPrint p = new IntPrint();
exp2(p).print(); // type-error
```

Here, we create a printing implementation `p` for integer expressions and try to use it on an expression `exp2` (see the definition below) defined for integer and boolean expressions. As expected, this fails to type-check.

The subtyping relation between terms is induced by the subtyping relation between object algebra interfaces. However, it follows the opposite direction: an extended term type (that is, with more constructors) is a *supertype* of a base term type. This subtyping relation is useful, for example, to build complex terms using an extended set of constructors from simpler terms:

```
<A> A exp(IntAlg<A> v) {
  return v.add(v.lit(3), v.lit(4));
}
<A> A exp2(IntBoolAlg<A> v) {
  return v.iff(v.bool(false),exp(v),v.lit(0));
}
```

In this case `exp` is a type of terms which can only be built using integer expressions, whereas `exp2` is type of terms which can use boolean expressions as well. Since terms for integer expressions are a subtype of terms for integer and boolean expressions, we can call `exp` in the definition of `exp2` with the object algebra argument `v` of type `IntBoolAlg<A>`.

Finally, note that there is an important difference to solutions to the EP using open classes [9]. In those solutions, there is a *single* expression type which is incrementally extended. So, once expressions are extended it becomes impossible to distinguish the extended expressions from more basic expressions. Thus, unlike a solution with object algebras, type distinctions between multiple variations of expressions are lost.

6 Multiple Types and Multi-sorted Object Algebras

In larger programs, it is often the case that we need multiple (potentially mutually) recursive types and operations evolving as a family. When the need for multiple types arises, we need to generalize from simple object algebras to multi-sorted object algebras. Multi-sorted object algebras also illustrate the relationship with the Abstract Factory pattern better, since this pattern is normally used with complex hierarchies with multiple types.

Multi-sorted object algebras are closely related to *family polymorphism* [12] which allow a variation EP where multiple types evolve as a family. Normally, without mechanisms like virtual types or classes, family polymorphism tends to be extremely heavyweight (and impractical) to encode [40]. However, as we shall see object algebras still scale well to the multiple type case.

```
interface StmtAlg<E, S> extends IntBoolAlg<E> {
  E var(String x);
  E assign(String x, E e);
  S expr(E e);
  S comp(S e1, S e2);
}
```

Fig. 8. Statements and expressions as a multi-sorted object algebra

6.1 Multiple Types

The need for multiple types appears, for example, when we want to have a language with expressions and statements. Figure 8 shows how to add statements to our little language of boolean and integers expressions. In order to introduce a new syntactic sort (statements) in the language, we need to add a new type parameter `S`. This corresponds effectively to having a multi-sorted (object) algebra, with `E` and `S` as the carrier types.[1]

As part of the statements object algebra interface we introduce two new forms of expressions: variables (`var`) and assignments (`assign`). We also introduce two forms of statements: sequential composition (`comp`) and liftings of expressions into statements (`expr`).

6.2 Evaluation of Statements: Algebras with Local State

Evaluation of statements is interesting for two reasons. Firstly it illustrates the definition of multi-sorted object algebras. Secondly it also illustrates an operation with local state: namely, the mapping between variables and values associated with those variables. If we would design the evaluation of statements using a more conventional OO style, with independent classes for variables and assignments, then we would have to coordinate the mapping of variables between those two classes. This could be done, for example, by explicit passing the variable mapping between those two classes. However, because this mapping is basically local to evaluation, this design is a bit unfortunate as it loses some encapsulation.

With the VISITOR pattern we could solve this problem more elegantly because we could create state that is local to an operation. Because object algebras also offer some of the same benefits as visitors, we can also exploit this design in our case. To do so, we use a design that is similar to retroactive interface implementations. This design is illustrated in Figure 9. Instead of creating individual classes, we use inner anonymous classes directly in the factory. The variable `map` keeps the mapping between variables and values. In the case of variables, we use `map` to retrieve the value associated with that variable. Assignments update the variable in the map with the value of the assigned expression. Composition evaluates the two expressions sequentially. Finally, `expr` simply returns the corresponding expression.

[1] Note that, more generally, we can encode sets of n (potentially mutually) recursive types by creating multi-sorted algebras with n type parameters (one for each type).

```
interface Stmt {
  void eval();
}
class StmtFactory extends IntBoolFactory implements StmtAlg<Exp, Stmt> {
  HashMap<String, Value> map = new HashMap<String, Value>();

  public Exp var(final String x) {
    return new Exp() {
      public Value eval() {
        return map.get(x);
      }};
  }
  public Exp assign(final String x, final Exp e) {
    return new Exp() {
      public Value eval() {
        Value v = e.eval();
        map.put(x, v);
        return v;
    }};
  }
  public Stmt comp(final Stmt s1, final Stmt s2) {
    return new Stmt() {
      public void eval() {
        s1.eval();
        s2.eval();
    }};
  }
  public Stmt expr(final Exp e) {
    return new Stmt() {
      public void eval() {
        e.eval();
    }};
}}
```

Fig. 9. An abstract factory for expressions and statements

Note that the object algebra in Figure 9 can also be interpreted as a *concrete builder* object from the Builder pattern [13]. The Builder pattern, like the Abstract Factory pattern, is a creational pattern. The main difference between the Builder pattern and the Abstract Factory pattern is that builders tend to have complex object construction processes. For example factories are typically stateless, while builders can maintain a state. In this case, `StmtFactory` is stateful, which would justify calling that class a concrete builder.

Client Code: The extended object algebra of expressions and statements can be used as before. For example we can create values for expressions and statements as follows:

```
<E,S> E exp(StmtAlg<E,S> v) {
  return v.assign("x", v.add(v.lit(3), v.lit(4)));
}
<E,S> S stmt(StmtAlg<E,S> v) {
```

```
interface BoolAlg<A> {
  A bool(boolean x);
  A iff(A b, A e1, A e2);
}

interface ExpIntBool<A> extends BoolAlg<A>, IntAlg<A> {}
```

Fig. 10. Composing algebra interfaces with interface inheritance

```
  return v.comp(v.expr(exp(v)), v.expr(v.var("x")));
}
```

Note that the syntactic restrictions which dictate where expressions and statements can occur are preserved. As such code like:

```
<E,S> S badStmt(StmtAlg<E,S> v) {
  return v.comp(exp(v), v.var("x")); //type-error
}
```

is rejected by the type-checker since it tries to use two expressions as arguments for sequential composition.

The evaluator is run as before: a factory is created, passed to `exp` and `stmt` and `eval` is invoked in the resulting objects.

```
StmtFactory factory = new StmtFactory();
exp(factory).eval();
stmt(factory).eval();
```

7 Modularity and Object Algebra Combinators

This section shows techniques to modularly define and compose independent components using object algebra combinator classes. One of the problems addressed in this section is how to achieve *independent extensibility* [50] in Java (Section 7.2). It is easy to have independent extensibility if a language supports traits [41] or mixin composition [2], but this is not as trivial in a language with single inheritance like Java. Another problem that is addressed in this section is the problem of defining retroactive implementations that depend on existing operations in the base classes or other retroactive implementations (Section 7.3).

7.1 Modular Combination of Algebra Interfaces

Interface inheritance can be used to combine algebra interfaces. For example, lets consider again the problem of developing boolean and integer expressions. In Figure 6 we opted to make the algebra interface for boolean expression extend that of integer expressions. However, there's nothing intrinsic to boolean expressions that depends on integer expressions. A more modular alternative implementation, shown in Figure 10, is to define integer and boolean algebras

```
class Union<A> implements IntBoolAlg<A> {
  BoolAlg<A> v1;
  IntAlg<A> v2;
  Union(BoolAlg<A> v1, IntAlg<A> v2) { this.v1 = v1; this.v2 = v2; }

  public A lit(int x)    { return v2.lit(x);   }
  public A add(A e1, A e2) { return v2.add(e1, e2); }
  public A bool(Boolean b) { return v1.bool(b); }
  public A iff(A e1, A e2, A e3) { return v1.iff(e1, e2, e3); }
}
```

Fig. 11. Composing operations with OO composition

as separate interfaces. Because most languages support multiple interface inheritance, this mechanism can compose the two algebra interfaces. The new interface `IntBoolAlg` illustrates this idea and shows how to compose `IntAlg` with `BoolAlg` through interface inheritance.

7.2 Modular Combination of Algebras

Unfortunately modular combinations of algebras themselves is not as easy as modular combination of algebra interfaces. The problem is that while languages like Java support multiple interface inheritance, they only support single implementation inheritance. As such we cannot use implementation inheritance in Java to compose two independent extensions.

However, OO composition offers an alternative way to combine modular extensions, although it takes some manual work to set up the composition. Fortunately it is possible to write fairly generic composition classes which allow composing different types of interpretations. At the high-level what we want is to define a combinator:

$$union \in V_1\ A \times V_2\ A \rightarrow (V_1 \otimes V_2)\ A$$

which takes two object algebras of type $V_1\ A$ and $V_2\ A$ and it returns the union of those algebras. In Java we can write *union* for two specific object algebras interfaces $V_1\ A$ and $V_2\ A$. Figure 11 illustrates the definition of *union* for the object algebras interfaces `BoolAlg` and `IntAlg`. We can use Java's multiple interface inheritance to approximate the union of two object algebras interfaces (that is the type-level operator $\otimes$). The actual implementation of the methods of `Union` is straightforward: each method simply delegates to the corresponding method in either `v1` or `v2`.

`Union` can be used to define the factory for boolean and arithmetic expressions from two independent extensions:

```
class IntBoolFactory2 extends Union<Exp> {
  IntBoolFactory2() { super(new BoolFactory(), new IntFactory()); }
}
```

Essentially all we have to do is to instantiate factories for boolean and integer expressions and invoke the constructor in `Union`. For retroactive implementations such as pretty printing we would proceed in the same way.

```
class Pair<A, B> {
  A a; B b;
  Pair(A a, B b) { this.a = a; this.b = b; }

  A a() { return a; }
  B b() { return b; }
}

class Combine<A, B> implements IntAlg<Pair<A, B>> {
  IntAlg<A> v1;
  IntAlg<B> v2;

  Combine(IntAlg<A> v1, IntAlg<B> v2) { this.v1 = v1;this.v2 = v2; }

  public Pair<A, B> lit(int x) {
    return new Pair<A, B>(v1.lit(x), v2.lit(x));
  }
  public Pair<A, B> add(Pair<A, B> e1, Pair<A, B> e2) {
    return new Pair<A, B>(v1.add(e1.a(), e2.a()), v2.add(e1.b(), e2.b()));
  }
}
```

Fig. 12. Combining operations in parallel

7.3 Combining Operations in Parallel

Sometimes it is useful to compose multiple operations together in such a way that they are executed in parallel to the same input. Abstractly speaking what we want is a combinator:

$$combine \in VA \times VB \rightarrow V(A \times B)$$

That is given two object algebras with types VA and VB we want to derive a third object algebra which combines the results of the two object algebras. This combinator is analogous to the `zip` function in functional programming, and it has been well-studied in the context of F-algebras [21].

Figure 12 shows how to define `combine` for integer expressions. Essentially, *combine* becomes an class (`Combine`) that is parametrized by two other object algebras `v1` and `v2`. The implementation of each method (`lit` and `add`) basically forwards the input to the corresponding cases in `v1` and `v2` and returns a pair with both results.

`Combine` is useful, for example, when we need to define operations that depend on multiple independent extensions. For example, consider adding some debugging information to the evaluator. In order to do this it is helpful to have a pretty printer. However, evaluation and pretty printing have been defined separately. By inheriting from `Combine` we can create a new class `Debug` that allows us to use evaluation and pretty printing at the same time.

```
class Debug extends Combine<Exp, IPrint> {
  Debug() { super(new IntFactory(), new IntPrint()); }
  Pair<Exp, IPrint> add(Pair<Exp, IPrint> e1, Pair<Exp, IPrint> e2) {
    System.out.println("The first expression " + e1.b().print() +
```

```
      " evaluates to " + e1.a().eval().toString());
   System.out.println("The second expression " + e2.b().print() +
      " evaluates to " + e2.a().eval().toString());
   return super.add(e1,e2);
}}
```

all we have to do is to invoke the constructor of the super class (`Combine`) with the integer expressions factory and pretty printer. Then to access the pretty printer and the evaluator, we just select the right component of the combined pair.

7.4 Some Final Notes on Extensibility

The attentive reader may have noticed two additional extensibility challenges that were left unaddressed. The first challenge is that the algebra class combinators that were just introduced are not extensible. The second challenge is that while expressions (and statements) are extensible the values computed by evaluation are not.

Both problems can be solved, but they require the most advanced use of generics in this paper: bounded polymorphism. However, we are still able to avoid F-bounded polymorphism since there is no need for recursive F-bounds.

We describe the key ideas of the solutions next, but refer the reader to our online implementation for the full code.

Extensible Algebra Combinators: Both `Union` and `Combine` cannot be easily extended. This is because these classes require concrete classes like `IntAlg` or `BoolAlg` in order to refer to the types of the object algebra parameters. It would be quite unfortunate if those algebra combinator classes could not be extended, because this would mean that each extension would have to create new algebra combinator classes from scratch.

To make such algebra combinator classes extensible we first observe that in `Union` and `Combine` there is no need to know about the concrete classes of the object algebra parameters. Rather only the upper bounds matter. Exploiting this observation we can define generalized versions of `Union` and `Combine` as follows:

```
class GUnion<A, V1 extends BoolAlg<A>, V2 extends IntAlg<A>> implements
    IntBoolAlg<A> {
 V1 v1; V2 v2;
 GUnion(V1 v1, V2 v2) { this.v1 = v1; this.v2 = v2; }
 ...
}
class GCombine<A, B, V1 extends IntAlg<A>, V2 extends IntAlg<B>>
        implements IntAlg<Pair<A, B>> {
 V1 v1; V2 v2;
 GCombine(V1 v1, V2 v2) { this.v1 = v1;this.v2 = v2; }
 ...
}
```

Unlike `Union` and `Combine` these classes allow extensibility because the bounds can be refined when the classes are extended. Therefore when defining combinators for extensions we can extend the classes `GUnion` and `GCombine` to inherit the cases for literals and addition.

Extensible Values: A similar idea is used to allow extensible values as well as extensible expressions.

```
interface IntVal<A> {
  A lit(int x);
}
interface IntExp<A> extends IntVal<A> {
  A add(A x, A y);
}
interface IntValue {
  int getInt();
}
class Eval<A extends IntValue,V extends IntVal<A>> implements IntExp<A> {
  protected V valFact;
  public Eval(V valFact) { this.valFact = valFact; }
  public A lit(int x) { return valFact.lit(x); }
  public A add(A x, A y) { return valFact.lit(x.getInt() + y.getInt()); }
}
```

Integer value factories are described by the `IntVal` algebra interface. There's a single constructor `lit`. The algebra interface for expressions then becomes an extension of `IntVal`. We also need an integer value interface (`IntValue`) for evaluation. With these 3 interfaces we can define an evaluation class (`Eval`) by parametrizing that class by a value factory (`valFact`). This factory is then used to avoid the use of concrete value constructors (as done in Figure 11). Again, because the bounds are refinable in extensions, this design allows extensibility of the value types.

8 Case Study

We have used this technique in implementing a new client model for invoking remote procedure calls (RCP), web services, and database clients (SQL). The system is called *batches* [22,48]. The system uses a custom scripting language to communicate batches of operations from clients to servers. The base object algebra of the system is defined in Figure 13. Some helper functions (which are technically redundant) are omitted for brevity.

There are currently five implementations of this signature. The first three implement direct evaluation of scripts, secure evaluation, and SQL translation, respectively.

The direct evaluation classes are similar to the classes defined in Section 4. The secure evaluation classes have the same basic structure, but carry additional state so that they can check the legality of each operation for the current user.

The SQL translator injects the algebra into a object hierarchy that has multiple mutually recursive translation functions, and also has additional SQL-specific

```
interface BatchFactory<E> {
  E Var(String name); // variable reference
  E Data(Object value); // simple constant (number, string, or date)
  E Fun(String var, E body);
  E Prim(Op op, List<E> args); // unary and binary operators
  E Prop(E base, String field); // field access
  E Assign(Op op, E target, E source); // assignment
  E Let(String var, E expression, E body); // control flow
  E If(E condition, E thenExp, E elseExp);
  E Loop(Op op, String var, E collection, E body);
  E Call(E target, String method, List<E> args); // method invocation
  E In(String location); // reading and writing forest
  E Out(String location, E expression);
}
```

Fig. 13. Batch script language abstract syntax

```
interface SQLTranslation {
  void toSQL(StringBuilder sb, List<Object> params, Forest data);
  Expression normalize(ISchema schema, SQLQuery query,
     Expression outerCond, Env env, NormType normType);
  SQLTable getTable();
  Expression invertPath(Expression e, Env env, boolean fromChild);
  SQLTable getTableNoJoins(Env env);
  SQLTable getBase(Env env);
  Expression withoutTransformations();
  Expression getTransformations(Expression base);
  Expression trimLast(Env env);
}
```

Fig. 14. Interface of mutually recursive methods used by the SQL Translation algebra

objects. The signature of this class is given in Figure 14. Using a traditional visitor approach, every one of these functions would have to be defined as a mutually recursive visitor class. With object-algebras, the SQL translation objects can call methods on sub-objects in the normal object-oriented style.

The final implementations are the most complex. They implement the partitioning mechanism required by the batch compiler. There are two parts, a partitioning algebra and a code generation algebra. The partition algebra extends the base algebra signature with additional node types, to represent code that does not belong to the batch. The methods, which are also mutually recursive, are listed in Figure 15. The partition system then creates batches, but then must *visit* the resulting objects to generate new code after the partition is complete. This is the one case where something like a traditional visitor is used. However, it is not used to create new operations. Instead it is used to *build* the new objects into a final code-generation algebra.

Subjectively this architecture allows the different subsystems (security, partitioning, and SQL translation) to be kept separate. Within each subsystem ordinary object-oriented dispatch is used. The main difference is that rather

```
interface PartitionFactory<E> extends BatchFactory<E> {
  E Other(Object external, E... subs);
  E DynamicCall(E target, String method, List<E> args);
  E Mobile(String type, Object obj, E exp);
}
```

Fig. 15. Extended script language interface used for partitioning algebra

than constructing generic operations, and then trying to create complex mutually recursive visitors that operate on the generic objects, the batch system creates specialized objects for each task.

9 Related Work

Throughout the paper we have already compared object algebras with several other related work. In this section we discuss additional related work.

Expression Problem in Java-like languages: Object algebras require only simple generics and work in languages like Java or C#. As far as we know Torgersen's [43] work on the EP presents the only solutions in the literature that can also work in those languages. He presents 4 solutions, but the first 3 solutions require advanced features like F-bounds or wildcards, while the last solution makes use of C# reflection mechanisms and it does not satisfy the (static) type-safety requirement of the expression problem. A drawback of Torgersen's first 3 solutions is that they require quite a bit of redundant code just for the purposes of satisfying the type-checker, and they are conceptually quite heavy. F-bounds and wildcards are notorious for being difficult to grasp for everyday programmers. An advantage of object algebras is that they are comparably lightweight on the amount of type annotations and they do not use those advanced features.

Before Torgersen's work, there have been attempts to provide solutions that work in Java-like languages. Wadler proposed a solution using generics to solve the expression problem [46], but he later found a subtle typing problem. Kim Bruce [3] proposed a solution to the expression problem using generics and *self-types*. However self-types are not a widely available feature and, as such, his solution does not work in most current mainstream OO languages. Finally, there have also been some other solutions that are not statically type-safe, but still allow extensibility [23,36,45].

Modular Visitors, Encodings of Datatypes and Embedded DSLs: There has been a considerable amount of work on modular visitors and related techniques in advanced programming languages like Haskell or Scala recently. This line of work is closely related to object algebras.

Hinze [19] was the first to point out that type classes provide a way to represent encodings of datatypes in Haskell. He exploited this fact to implement a generic programming library. Inspired by Hinze's work, Oliveira and Gibbons [32] have shown general patterns for those techniques and used them to

several other applications. In following work Oliveira et al. showed that variants of these type-class based encodings are extensible and can be used to solve the expression problem [33]. Later work by Carrete et al. [7] and Hofer et al. [20] (in Scala) popularized those techniques for defining well-typed interpreters and embedded DSLs. While all that work is closely related to object algebras, those techniques require significant advanced language features not available in mainstream OO languages like Java. A source of additional complexity in this line of work is that most documented applications use encodings of *generalized* algebraic datatypes [37]. Even with our simplified techniques, it is not possible to define Church encodings of generalized algebraic datatypes in Java as this requires type-constructor polymorphism [39,28].

The relationship between the VISITOR pattern and Church encodings has been folklore in the type-theory community. Buchlovsky and Thielecke [5] were the first to precisely document that relationship. The link between the type class based encodings, the VISITOR pattern, encodings of datatypes and the extensibility of such encodings was further developed by Oliveira [30,35]. In later work, Oliveira [31] generalized and showed how to apply his results on extensibility to two variations of the VISITOR pattern. Still the Scala implementation of that work used advanced features including type constructor polymorphism, variance annotations, self-type annotations and mixins. None of these are available in Java. A key insight of our work is that by avoiding `accept` methods and using plain object algebras instead we can get most of the benefits of modular visitors in Java. Furthermore, unlike visitors, object algebras support retroactive implementations without requiring `accept` methods.

Another Haskell solution to the expression problem is based on folds (and F-algebras) [11,42]. This solution also requires advanced features and it does not translate well to object-oriented programming because most OO languages do not have native support for sums-of-products, which are needed in that solution.

Finally Zenger and Odersky [50] proposed a solution to the expression problem in Scala using virtual types [4] and the *open class pattern* [27]. As most other solutions discussed here, this solution is quite heavyweight in terms of language features and it requires a lot of type annotations and manual composition code.

Language-Based Solutions to the Expression Problem: Several programming language features such as *multi-methods* [8], *open classes* [9], *virtual classes* [25,29], *virtual types* [4], *units* [27], *polymorphic variants* [14] and others [49,24,47] are aimed at solving problems related to the expression problem. The main advantage of most of these approaches is that solutions to the expression problem can be expressed quite naturally. In contrast, solutions that instead exploit general programming language features (like generics or type classes) are commonly criticized for being heavyweight, hard to use and requiring sophisticated features. Our work shows that is possible to significantly reduce such complexity by using object algebras. There is still some price to pay in terms of indirection, but compared to other approaches using general programming language features this is a relatively small cost: it is low enough that object algebras are useful in practice. The main advantage of object algebras over language-based approaches is that

object algebras do not require a new language or language extension: they can be used in any languages that support a simple form of generics.

Visitor Combinators and Functional Interpretations of Design Patterns: There has been some work on visitor combinators for offering better traversal control. Visser [45] presented a number of combinators that can express interesting traversal strategies like bottom-up, top-down or sequential composition of visitors. This work is related to our algebra combinators in Section 7. However a difference is that we use *functional* style object algebras whereas Visser uses *imperative* style visitors. As part of our future work we would like to explore more algebra combinators and develop a small algebra of combinators for object algebras.

Gibbons [16] has proposed functional interpretations for various design patterns. In particular he suggested that the VISITOR and the BUILDER patterns are closely related to folds in functional programming. Our work supports this idea and extends it, suggesting that the ABSTRACT FACTORY pattern is also also part of the functional interpretation as folds.

10 Conclusion

This paper presents a new solution to the expression problem based on object algebras. This solution is interesting because it is extremely lightweight in terms of required language features; has a low conceptual overhead for programmers; and it scales well with respect to other challenges related to the expression problem.

Object algebras promote a factory-oriented programming style where concrete constructors are avoided. This programming style has some overhead over a conventional object-oriented programming style, but it also offers several advantages in terms of extensibility. In comparison with the VISITOR pattern, object algebras retain most of the advantages and additionally they support extensibility and do not require `accept` methods. As such object algebras can provide retroactive implementations even when the original source code is not available.

Although this paper shows that object algebras can be encoded in languages like Java, programming language extensions are still useful. With additional language support we expect programming with object algebras to be even more convenient. For example programming language extensions can be useful to manage factories better, or to automatically provide composition operators for object algebras. This is something we would like to explore in future work.

Acknowledgements. We are grateful to Gavin Bierman, Alex Loh, Matthew Parkinson, Tom Schrijvers, Tijs Van der Storm, Tarmo Uustalu and the anonymous reviewers for various discussions, comments and suggestions about this work. This research was funded by the UT Austin-Portugal Colab Program and by Singapore Ministry of Education research grant MOE2010-T2-2-073.

References

1. Bierman, G.M., Meijer, E., Torgersen, M.: Lost in translation: formalizing proposed extensions to C#. In: OOPSLA 2007 (2007)
2. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA/ECOOP 1990 (1990)
3. Bruce, K.: Some challenging typing issues in object-oriented languages: Extended abstract. Electronic Notes in Theoretical Computer Science 82(8), 1–29 (2003)
4. Bruce, K.B., Odersky, M., Wadler, P.: A Statically Safe Alternative to Virtual Types. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 523–549. Springer, Heidelberg (1998)
5. Buchlovsky, P., Thielecke, H.: A type-theoretic reconstruction of the visitor pattern. Electronic Notes in Theoretical Computer Science 155(0), 309–329 (2006)
6. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: FPCA 1989 (1989)
7. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19, 509–543 (2009)
8. Chambers, C., Leavens, G.T.: Typechecking and modules for multimethods. ACM Trans. Program. Lang. Syst. 17, 805–843 (1995)
9. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for java. In: OOPSLA 2000 (2000)
10. Cook, W.R.: Object-Oriented Programming Versus Abstract Data Types. In: de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1990. LNCS, vol. 489, pp. 151–178. Springer, Heidelberg (1991)
11. Duponcheel, L.: Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters (1995), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.7093
12. Ernst, E.: Family Polymorphism. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addisson-Wesley professional computing series. Addisson-Wesley (1994)
14. Garrigue, J.: Programming with polymorphic variants (1998)
15. Ghani, N., Uustalu, T., Vene, V.: Build, Augment and Destroy, Universally. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 327–347. Springer, Heidelberg (2004)
16. Gibbons, J.: Design patterns as higher-order datatype-generic programs. In: WGP 2006 (2006)
17. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press (1989)
18. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica (1978)
19. Hinze, R.: Generics for the masses. Journal of Functional Programming 16(4-5), 451–483 (2006)
20. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of DSLs. In: GPCE 2008 (2008)
21. Hoogendijk, P., Backhouse, R.: When Do Datatypes Commute? In: Moggi, E., Rosolini, G. (eds.) CTCS 1997. LNCS, vol. 1290, pp. 242–260. Springer, Heidelberg (1997)
22. Ibrahim, A., Jiao, Y., Tilevich, E., Cook, W.R.: Remote Batch Invocation for Compositional Object Services. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 595–617. Springer, Heidelberg (2009)

23. Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing Object-Oriented and Functional Design to Promote Re-Use. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 91–113. Springer, Heidelberg (1998)
24. Löh, A., Hinze, R.: Open data types and open functions. In: PPDP 2006 (2006)
25. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA 1989 (1989)
26. Malcolm, G.: Algebraic Data Types and Program Transformation. Ph.D. thesis, Rijksuniversiteit Groningen (September 1990)
27. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: new-age components for old-fashioned java. In: OOPSLA 2001 (2001)
28. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: OOPSLA 2008 (2008)
29. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA 2006 (2006)
30. Oliveira, B.C.d.S.: Genericity, extensibility and type-safety in the Visitor pattern. Ph.D. thesis, Oxford University Computing Laboratory (2007)
31. Oliveira, B.C.d.S.: Modular Visitor Components. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 269–293. Springer, Heidelberg (2009)
32. Oliveira, B.C.d.S., Gibbons, J.: Typecase: a design pattern for type-indexed functions. In: Haskell 2005 (2005)
33. Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Trends in Functional Programming (2006)
34. Oliveira, B.C.d.S., Moors, A., Odersky, M.: Type classes as objects and implicits. In: OOPSLA 2010 (2010)
35. Oliveira, B.C.d.S., Wang, M., Gibbons, J.: The visitor pattern as a reusable, generic, type-safe component. In: OOPSLA 2008 (2008)
36. Palsberg, J., Jay, C.B.: The essence of the visitor pattern. In: COMPSAC 1998 (1998)
37. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP 2006 (2006)
38. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In: Schuman, S.A. (ed.) New Directions in Algorithmic Languages, pp. 157–168 (1975)
39. Reynolds, J.C.: Towards a Theory of Type Structure. In: Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 408–425. Springer, Heidelberg (1974)
40. Saito, C., Igarashi, A.: The essence of lightweight family polymorphism. Journal of Object Technology, 67–99 (2008)
41. Scharli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable Units of Behaviour. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
42. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (2008)
43. Torgersen, M.: The Expression Problem Revisited – Four New Solutions Using Generics. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
44. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: SAC 2004 (2004)
45. Visser, J.: Visitor combination and traversal control. In: OOPSLA 2001 (2001)
46. Wadler, P.: The Expression Problem. Email (November 1998), discussion on the Java Genericity mailing list

47. Wehr, S., Thiemann, P.: JavaGI: The interaction of type classes with interfaces and inheritance. ACM Trans. Program. Lang. Syst. 33 (July 2011)
48. Wiedermann, B., Cook, W.R.: Remote batch invocation for SQL databases. In: The 13th International Symposium on Database Programming Languages, DBPL (2011)
49. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: ICFP 2001 (2001)
50. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL 2005 (2005)

Extensions during Software Evolution: Do Objects Meet Their Promise?

Romain Robbes, David Röthlisberger, and Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile, Chile
http://pleiad.cl

Abstract. As software evolves, data types have to be extended, possibly with new data variants or new operations. Object-oriented design is well-known to support data extensions well. In fact, most popular books showcase data extensions to illustrate how objects adequately support software evolution. Conversely, operation extensions are typically better supported by a functional design. A large body of programming language research has been devoted to the challenge of properly supporting both kinds of extensions.

While this challenge is well-known from a language design standpoint, it has not been studied empirically. We perform such a study on a large sample of Smalltalk projects (over half a billion lines of code) and their evolution over more than 100,000 committed changes.

Our study of extensions during software evolution finds that extensions are indeed prevalent evolution tasks, and that both kinds of extensions are equally common in object-oriented software. We also discuss findings about the evolution of the kinds of extensions over time, and about the viability of the Visitor pattern as an object-oriented solution to operation extensions. This study suggests that object-oriented design alone is not sufficient, and that practical support for both kinds of program decomposition approaches are in fact needed, either by the programming language or by the development environment.

1 Introduction

Lehman's laws of software evolution [13] tell us that software systems must continuously adapt, or become progressively less useful to their users. Over time, new functionality is added to software systems. Inevitably, some functionality needs to extend existing system components. Depending on the programming paradigm used, different extensions have different consequences.

Extensions can happen along two dimensions: new data variants, or new operations. Object-oriented programming is well-known for seamlessly supporting extensibility of data variants, by introducing new kinds of objects. In contrast, the functional design approach [12]—where the variants of a data type are processed by case-analyzing procedures—is better suited to support additions of

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 28–52, 2012.

new operations, by introducing new procedures. Conversely, supporting new operations for objects requires modifying all object definitions to add new methods, and adding new data variants in the functional approach implies modifying all existing procedures to handle the new cases.

This complementarity between data types and "procedural data values" (objects) dates back to the work of Reynolds in the 1970s [18] and has been described by other researchers since then (*e.g.* [5,12,26]). Supporting both forms of extensions appropriately is a challenge that has a strong practical relevance, because the choice of a programming paradigm (or design approach) greatly influences the kind of extension that is supported in a localized manner, without modifying existing code. For instance, choosing an object-oriented decomposition to implement a system whose evolution predominantly involves operation extensions is like using a hammer to paint a wall: possible, but painful. The object-oriented programming community has in fact designed a solution to handle operation extensions, called the Visitor design pattern [7]. A visitor makes it possible to turn an operation extension scenario into a data extension scenario. But once adopted, the Visitor pattern complicates data extensions. Many programming language constructs have been proposed in order to support both dimensions of extension in a modular (and type safe) manner (*e.g.* [12,15,24,27]).

As a matter of fact, the literature on object-oriented programming very often illustrates the superiority of objects in dealing with software evolution by showcasing data extension scenarios (see Booch [4] and Shalloway Trott [22] for two popular books). However, there is no empirical data on how frequently such extensions do occur, nor is there evidence that data extensions are significantly more common than operation extensions, even in object-oriented software.

The extensibility challenge can be looked at both from the point of view of the implementers of a system—the kinds of extensions that have to be dealt with in the evolution of the system—and from the point of view of black-box third-party extensions (the latter is usually seen as the extensibility/expression problem stricto sensu [26,24]). This work is concerned with the first part of the question. We study the evolution of open-source object-oriented projects through their commit history, looking at how the implementers of a project add new data variants and operations to their class hierarchies as the system evolves. Even if there is no strong impediment to change existing code in this setting, being able to express these extensions modularly does matter; it is well-known that most of the costs of software development are in maintenance and evolution, not in initial development [6].

Concretely, we seek to answer the following research questions:

Q1: Are extensions prevalent in practice? Looking at the evolution of software, is it really the case that new data variants and operations are frequently added? Or are other kinds of changes (*e.g.* changing the implementation of a method) much more common as to render the point moot?

Q2: Are data extensions more common than operation extensions? If object-oriented programming is really superior in dealing with extensible software, object-oriented projects should showcase a far greater number of

data extension cases. Is it really the case? Or conversely, are there much more operation extensions, suggesting that another programming abstraction would be more adequate? Or, are both kinds of extensions similarly important in practice?

Q3: How do extensions occur over time? Over the lifetime of an object-oriented system, do both kinds of extensions manifest regularly? Or are unanticipated design decisions leading to more problematic extension cases as the system ages?

Q4: Is the Visitor pattern a suitable solution? How much is the Visitor pattern used in practice? In cases where it is adopted, are its benefits clearly observable? Are visitor and visited hierarchies more stable than others?

By observing the evolution of a large number of open source Smalltalk projects, this paper presents elements of answers to these questions. Note that because Smalltalk is a dynamically-typed language, this study does not answer these questions in a typed context. Whether or not static typing has an influence on extensions during software evolution is an open question that future studies should address. Also, because Smalltalk is an object-oriented language, we get to observe how object programmers actually benefit (or not) from working in that paradigm. Studies based on other languages, including those that natively support both decomposition approaches, would be needed to answer the research questions above in general. This study is therefore a first step towards providing substance to the long-running debate that takes place in the programming language research community about different forms of data abstractions.

Structure of the Paper. Section 2 briefly reviews background and related work. Section 3 describes the experimental setup, explaining how the data was collected and processed. The next four sections report our findings related to the four research questions stated above. Section 8 discusses threats to the validity of this study, and Section 9 concludes.

2 Background and Related Work

We first explain the different kinds of extensions and how to deal with then in object-oriented programming, including the Visitor design pattern. We then review related studies of object-oriented programming practice.

2.1 Extensibility in OOP

Consider the object-oriented design of a simple programming language of arithmetic expressions (Figure 1a)[1]. Expression subclasses `Num` and `Add` implement their own `eval`uation method. A first kind of extension is *data extension*, which consists in adding new data variants; in that case, a new kind of expression

[1] Anticipating the fact that we study Smalltalk code, we present the example in a dynamically-typed class-based setting, using inheritance to define data variants.

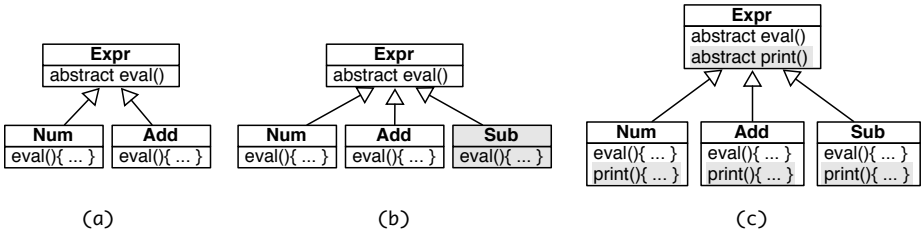


Fig. 1. A class hierarchy (a) and two extensions: data extension (b), and operation extension (c). Changes are highlighted in gray.

(Figure 1b). Note how the object paradigm makes this extension localized: it is enough to add a new subclass. A second kind of extension is *operation extension*, *e.g.* extending the protocol of expressions, such that they can also be pretty-`print`ed. The object-oriented decomposition is much less suited for this kind of extension, which requires invasive and non-localized modifications of existing classes (Figure 1c).

The Visitor design pattern [7] is the standard way to handle operation extensions in a modular manner in object-oriented programming. It consists of preparing the hierarchy to extend so that it accepts visitor objects (*e.g.* add a method `accept` on each class of the `Expr` hierarchy). A separate hierarchy of visitors is then defined, each for its own operation (*e.g.* `PrintVisitor` extends from `ExprVisitor`). Adding a new operation on the hierarchy is now expressed as adding a new visitor subclass (*e.g.* `TypeVisitor`). Note that applying the Visitor pattern increases the complexity of the system, and that adding a new data variant in the visited hierarchy (*e.g.* `Mult`) implies extending all the visitors.

2.2 Related Work

As far as we are aware, there are no empirical studies of the prevalence of extensions during software evolution, nor on comparing the kinds of extensions (data vs. operation) that happen in real world projects. There are however several related studies of characteristics of source code, class hierarchies, and their evolution.

Gîrba *et al.* define a visualization of class hierarchies that incorporates evolutionary metrics, such as age of class, age of inheritance links *etc.* [8]. Based on a study of two open-source systems, they identify several visual patterns to characterize the evolution of the hierarchies. The patterns are however coarse as the unit of granularity is the class, and are aimed to answer general evolution questions, such as the distribution of changes across hierarchies.

A study by van Rysselberghe and Demeyer analyzed hierarchy changes on two Java systems [20]. The exploratory study led to the formulation of 7 hypotheses to be investigated, such as "Hierarchy changes are likely to insert an additional abstraction between the old parent and the center class" and "Inheritance is only rarely replaced by composition". Due to its limited extent this study however only

hinted at the answer to the hypotheses; its findings need to be confirmed by a larger-scale study.

Baxter *et al.* performed an empirical study on 16 releases of several Java systems, in order to investigate the distribution of several metrics and whether those followed power laws [3]. Later, Tempero *et al.* focused on the use of inheritance in Java software, using the same corpus (expanded to 93 programs), and a suite of 23 metrics [23]; they found a larger amount of inheritance than they expected: around three-quarters of classes used inheritance (for half of the applications in the corpus). A large-scale survey of programmers by Gorscheck *et al.* [9] found a lack of consensus on what the size of classes and depth of hierarchies should be. A recent large-scale study (2,080 Java programs, found on Sourceforge) by Grechanik *et al.* formulated 32 research questions [10]. Of those, several were related to class hierarchies. They found that almost 50% of the classes are written without using inheritance, and that 71% of the hierarchies had a depth of one. These findings differ somewhat from the ones of Tempero, who found a higher usage of inheritance. However, the metrics used in both studies differ, so comparison is difficult. All of these studies investigate a large number of research questions—trading depth for breadth—while we focus on the handful of questions that allow us to characterize extensions during the evolution of object-oriented software.

Finally, Aversano *et al.* studied the evolution of several design patterns, including the Visitor pattern, on 3 software systems [2]. They found that classes involved in the Visitor pattern were among the most changed in one of the systems (Eclipse JDT), but that the changes were mostly in the visitor hierarchy, not in the visited hierarchy. The study considers design patterns in general, and is focused on three systems only.

3 Experimental Setup

3.1 Data Collection

We analyze a large extract of the *Squeaksource*[2] repository for Smalltalk projects written in either Squeak or Pharo (a fork of Squeak). Squeaksource is the foundation for the software ecosystem that the Squeak and Pharo community have built over the years. The majority of Squeak and Pharo developers use Squeaksource as their primary source code repository, making it a nearly complete view of the Squeak and Pharo software ecosystem. The Squeaksource extract we analyze spans 8 years and involves approximately 2,500 projects consisting of more than 95,000 unique classes. Summing all versions of all projects yields nearly 600 million lines of code. Over the course of these 8 years, more than 2,300 developers committed around 110,000 changes to Squeaksource.

To version their source code in Squeaksource, developers use the versioning system *Monticello*. When committing a new version of a project, Monticello stores a snapshot of the entire committed package, without computing the delta

[2] http://www.squeaksource.com

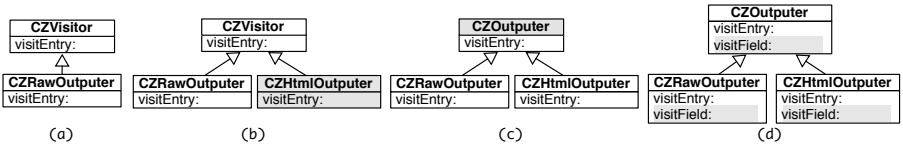


Fig. 2. The Citezen visitor hierarchy: (a) initial version, (b) after a data extension, (c) after renaming the root class, and (d) after an operation extension

to the previous version. Squeaksource is hence a large filesystem directory. With each commit, meta-information is recorded. Monticello versions code at the level of packages, classes, and methods, not at the level of files and lines of code.

To actually analyze the Squeaksource repository we use the Ecco model [19]. Ecco is a lightweight representation of software systems and their versions in an ecosystem. The main unit of abstraction is the system (or software project). For each pair of successive versions of a system, Ecco only keeps the changes between the versions. These changes consist of sets of additions, modifications, and removals of classes and methods in the system. Meta-information such as author, timestamp, and links to one or more ancestors and successors versions are maintained as well. Ecco allows us to effectively and efficiently process and analyze the large set of changes (approx. 13GB of compressed source code) we extracted from Squeaksource.

3.2 Data Processing

Before analyzing extension scenarios in the code base, we process the changes from Squeaksource in various steps, described below.

Extension Detection. We limit our analysis to the granularity of methods. We do not look into the source code of methods, but stop at the method boundary. Moreover, we only study the additions of classes and methods, but not their modifications. The kinds of extensions can be well quantified by keeping track of additions of classes and methods, since a data extension corresponds to the addition of one or more classes to a hierarchy and an operation extension to the addition of a new method to several classes of a hierarchy.

To detect operation and data extensions, we track the evolution of each class hierarchy in a software project. A real example of such a hierarchy evolution is depicted in Figure 2, which shows the visitor hierarchy of `Citezen`, an application for managing bibtex files on the web. If in a particular change a new class is added to a hierarchy, we consider the addition to be a data extension[3] (shown in Figure 2b where class `CZHtmlOutputer` has been added). The addition of a method with the same name to a least two classes of a hierarchy in a particular change is considered to be an operation extension (*e.g.* Figure 2d, where method `visitField:` is introduced). If a change adds only one method to a class hierarchy, but previous or subsequent changes add methods with that name to the

[3] Adding a new subclass of `Object` is not considered a data extension.

same hierarchy, this single method addition is also considered to be part of an operation extension.

In the case where several classes containing methods with the same name are added to the same hierarchy in the same change, these added methods could actually be detected as operation extensions. For a data extension, we however consider all methods added in the new classes to belong to this data extension, thus no operation extensions are identified in such a scenario.

The root class of any hierarchy can be renamed during the lifetime of a project (*e.g.* root class `CZVisitor` is renamed to `CZOutputer` in Figure 2c). In Monticello, renaming an entity means removing the entity with the old name and adding a new entity with the new name. As we can hence not directly determine a rename of a root class in the changes, we employ an algorithm that tests for every removed root class whether any class newly added in the same change might actually be the renamed version of this root class. For this we compare the set of methods of the removed class with the one of the newly-added class (subclasses of the new and old class are not compared to make the algorithm independent of possible renames to subclasses occurring in the same change). If these sets overlap for at least 80% of the methods, we consider this change as a rename and exchange the old root of the hierarchy with the newly-added class.

Extension Weighting. To estimate the effort to realize an extension, it is not enough to compare the number of operation extensions with the number of data extensions. The former is adding just methods, while the latter is adding an entire class to a system. For this reason, we weight a data extension with the number of methods with which the class has been added to the system to obtain a measure for the effort needed to perform an extension. This allows us to compare operation and data extensions in terms of effort, while in the unweighted case we compare the frequency of the two kinds of extensions.

Visitor Detection. To detect occurrences of the Visitor pattern and extensions to them, we search for methods whose name is starting with `accept` or `visit`. What follows this prefix is usually the name of the class being visited, *e.g.* `visitField:` typically accepts an instance of class `Field` (or subclasses). The visitor hierarchy is the class hierarchy in which one or more methods following this name pattern are located, while the visited hierarchy is the hierarchy containing the visited class (*e.g.* `Field`). We also support the case when a visitor is visiting various hierarchies, or when a visited hierarchy is visited by several independent visitors.

Aggregation. Beyond class hierarchies, we are also interested in how our analysis translates to the level of *projects*. Recent work by Posnett *et al.* shows that findings at one level of abstraction do not necessarily translate to finer or coarser levels—a phenomenon known as the *ecological fallacy* [17]. For our study we expect that the proportion of projects featuring extensions is higher than the same proportion for class hierarchies. Since the extensions could also be concentrated on a few, possibly large projects, the project level analysis is important to reveal how extensions are distributed over the projects.

Classification. To ease the analysis of the data, we classify the changes, that is, the commits to the projects in three categories: (i) initial, (ii) large, and (iii) selected commits. (i) The first commit to a project reflects the initial development of a project. (ii) Large commits consist of more than 50 added classes and methods. Note that initial commits are often also large commits. (iii) Selected commits are all commits neither classified as initial nor large. This classification is necessary because initial commits carry no change information, and large commits can hardly be meaningfully analyzed because they contain too many changes and are therefore considered as noise [28].

We also classify class hierarchies and projects in two categories: (i) all and (ii) large hierarchies or projects. A large hierarchy has a size of more than five classes. A large project is one with more than 50 classes. This classification is interesting because the impact of an operation extension is arguably more critical in large cases.

Filtering. A large and publicly accessible repository like Squeaksource typically also contains many toy or abandoned projects that would add undesired noise to our analysis. Hence we only take into account class hierarchies that have been changed at least five times and that contain at least two classes (one root and one subclass). Except for the first measurement of Section 4, we only analyze selected commits.

3.3 Basic Statistics in Squeaksource

Processing the dataset as discussed gives us the following information to be analyzed in detail in subsequent sections: We start with 111,071 commits; of those, 10,718 commits are classified as *large* or *initial commits*, leaving us with 100,353 *selected* commits. The 95,662 classes are organized in 48,595 hierarchies. Of those, 20,046 have more than one class. This means that 28,550 of the 95,662 classes (29.84%) do not use inheritance, a figure that concords with that reported by Tempero *et al.* [23]. Out of these hierarchies, we select 10,390 satisfying our thresholds of size (at least 2 classes) and activity (at least 5 changes); these are the focus of our analysis. Of these 10,390 class hierarchies, 2,879 have at least an operation or a data extension in selected commits. Also, 2,360 of these 10,390 class hierarchies are classified as large (more than 5 classes). We analyze 2505 projects, of which 569 are classified as large (more than 50 classes); 1036 of the projects feature either operation or data extensions in selected commits.

In a single commit, the largest operation extension we found added 40 methods to the hierarchy, whereas 36 classes were added to the same hierarchy in a single commit. This excludes large and initial commits, including several legitimate operation extensions. These large values lead us to investigate the distribution of the metrics.

Distribution of Metrics. Figure 3 shows the distribution of our metrics of interest across projects and class hierarchies. None of the distribution follows the characteristic "bell shape" of a normal distribution. Instead, the overwhelming majority

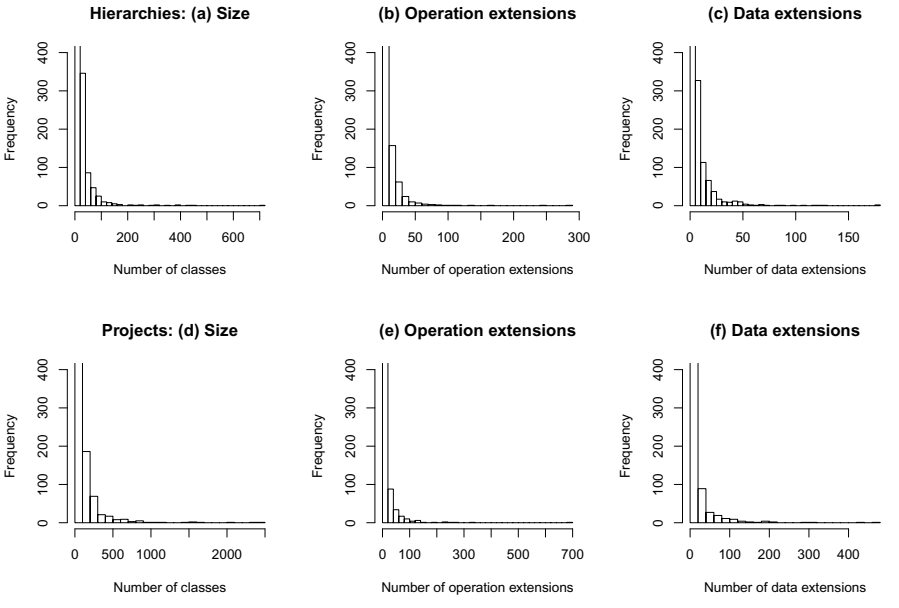


Fig. 3. Histograms showing the distribution of size and extension metrics across hierarchies and projects. Note that the first bar is cut as it would be too tall otherwise.

of observations has very low metric values, and a minority has high values. This observation and the presence of outliers on the tail end of the distributions, leads us to use robust descriptors when characterizing the distributions, *i.e.* the median instead of the mean, and boxplots (showing percentiles) as a visual summary of the distributions. In Section 5.2 we discuss which statistical tests can be used to analyze the data even though its distribution strongly departs from normality and thus breaks the assumptions of most parametric tests.

4 Are Extensions Prevalent?

We first estimate the prevalence of extensions by looking at the frequency of extension changes in commits. In a second step, we study the frequency of extensions in hierarchies and projects.

4.1 Frequency of Extensions in Commits

Intuitively, the frequency of extension events across commits tells us how often developers need to perform extensions over time: if extensions are extremely rare, then the challenge of dealing with both kinds of extensions is interesting from a theoretical standpoint, but has little practical impact. Figure 4 shows the proportion of commits featuring operation and data extensions versus commits that

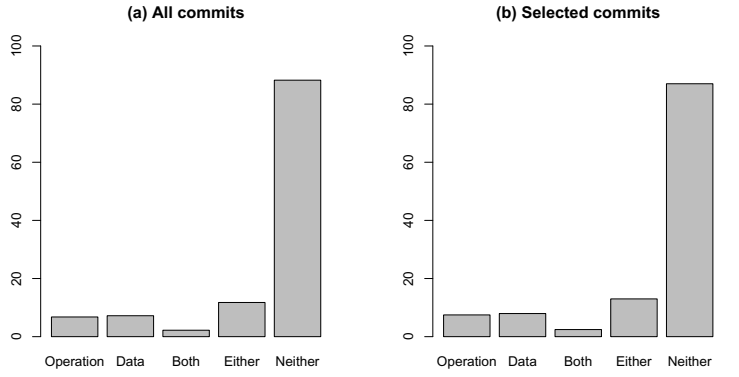


Fig. 4. Percentages of commits featuring extensions. (a) all commits; (b) selected commits.

feature neither of these[4]. Of the 111,071 commits we analyzed, 13,063 (11.76%) feature either an operation or a data extension. If we only consider selected commits (that is, we filter out both large and initial commits), the proportion rises to 12.99%. This means that in more than 1/8 of the commits, developers perform either a data or an operation extension in the system they work on.

An extension is problematic in an object-oriented program as soon as an operation extension is needed. We can see that operation extensions occur in 6.77% of all commits (7.48% of selected commits).

4.2 Frequency of Extensions in Class Hierarchies and Projects

To view the problem from another angle, we also measure the proportion of hierarchies that feature extensions at any given point in their life. This gives the proportion of hierarchies for which a developer will be expected to perform extensions. Figure 5 shows the proportions of hierarchies featuring at least one extension during their lifetime, versus hierarchies that do not. Out of the 10,390 hierarchies we observed, 27.70% (2,879) become subject of extensions sooner or later. Clearly, a large portion of hierarchies need refinements over time. Importantly, 19.35% of all class hierarchies are subject to operation extensions, which are not modularly supported by objects.

Intuitively, extensions are more problematic for larger hierarchies, where the complexity is higher. We measured the proportion of large hierarchies that are subject to extensions. We find that an overwhelming majority (1,883 out of 2,360, *i.e.* 79.81%) of these large hierarchies feature extensions. Also, more than 62.48% of these hierarchies are subject to operation extensions. Across large, more complex hierarchies, the modularity issue to express extensions is no longer a minority case; it is the norm.

[4] We discuss the relative prevalence of both kinds of extensions in Section 5.

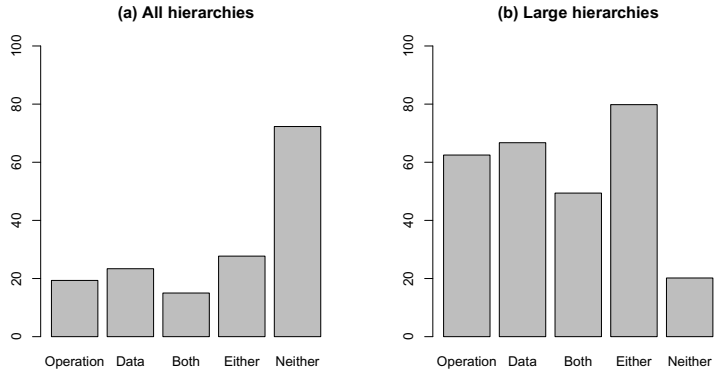


Fig. 5. Percentages of hierarchies featuring extensions. (a) all hierarchies; (b) large hierarchies only (5 or more classes).

For space reasons, we do not provide graphs for projects featuring extensions. In brief, 41% of all projects feature extensions (data extensions: 37.16%; operation extensions: 33.35%; both: 29.06%). Almost all (95%) large projects (*i.e.* projects with more than 50 classes) have to deal with extensions (data extensions: 87.07 %; operation extensions: 89.11%; both: 81.63%).

4.3 Executive Summary

Extensions regularly occur in practice: one out of eight commits (13%) features an extension. Further, a fifth of all class hierarchies have to be extended with new operations; this rate increases to over 60% for large hierarchies. We can conclude that developers are often confronted with extensions that are not modularly supported by object-oriented design. Moreover, for large hierarchies—where one can suppose the impact is more severe—the problem is all the more prevalent.

Far from being a theoretical curiosity, properly supporting both kinds of extensions is of practical concern for software developers, and hence effectively deserves the attention of the community.

5 Comparing Data and Operation Extensions

Having established that extensions are prevalent, we now focus on the distribution of the extension cases across the two categories of extensions. Underlying the research question is the intuition that if the object-oriented paradigm is well-suited for most kinds of evolutions, we expect data extensions to be *much more* common than operation extensions.

5.1 Frequency of Both Kinds of Events

We have already seen in the previous section that both kinds of extensions happen in practice. Looking back at Figure 4 and Figure 5, we notice that both

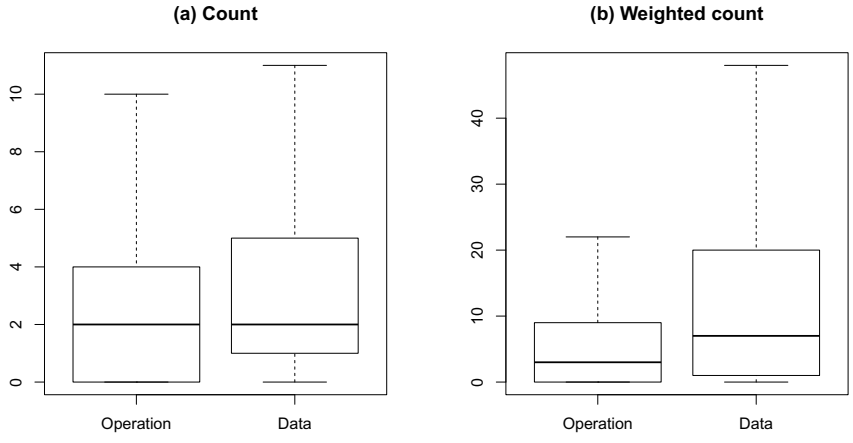


Fig. 6. Boxplots of distribution of extensions for hierarchies featuring them. (a) unweighted; (b) weighted.

types of extensions happen with a somewhat similar frequency (*i.e.* 7–8% of all commits, 60–65% of large hierarchies, etc.) and routinely overlap[5]. This gives us a first impression that operation extensions are actually *not* uncommon; rather, they seem to occur with relatively the same frequency as data extensions.

To investigate the problem more closely, we look at the distributions of both kinds of events, for the subset of hierarchies which experience these events. In order to evaluate the problem beyond frequency, we also look at the distributions of the weighted coefficients we introduced earlier—where the weight of an extension is defined by the number of methods it contains—, which gives us a more accurate metric of the effort involved in each kind of extension.

Figure 6 shows the distributions of both kinds of events as box-and-whiskers plots, for both unweighted—to evaluate frequency—and weighted—to evaluate effort—distributions, for the 27% of hierarchies that feature at least one of the two events during their life time. The thick line across each box represents the median value of the distribution, the boxes delimit the interquartile range (the 50% of the values that are between the 25th and the 75th percentile), while the whiskers depict the 5th and the 95th percentile; outliers are not displayed.

If only frequency is considered (Figure 6a), we see that in both cases, the median value (2) is identical. This tells us that the distributions are very similar in terms of frequency. The impression is reinforced by the significant overlap of the boxes. All in all, both kinds of extensions seem to happen with the same regularity, with data extensions being only slightly more common.

[5] Cases where both kinds of extensions overlap in the same hierarchy are especially interesting because they correspond to scenarios that no single data abstraction mechanism would be able to handle properly.

If we consider effort (Figure 6b), a different picture emerges. The median effort in introducing data extensions is higher (7 methods) than the effort involved in introducing operation extensions (3 methods). However, the boxes still overlap significantly: the 75th percentile of operation extensions is higher than the median of data extensions. If operation extensions need more effort, the difference is not so high that one can ignore operation extensions altogether.

Due to space limitations, we do not provide the graphs for commits and projects. These however feature the same pattern of an extremely large overlap in the unweighted case, and a still large overlap in the weighted case—with the upper quartile of weighted operation extensions above the median of weighted data extensions. For commits, the weighted data extension median is 2, while the weighted operation extension's 3rd quartile is 3; for projects, we have 20 and 25, respectively.

5.2 Quantifying the Difference between Kinds of Extension

A visual inspection of the distributions of extensions shows that the distribution of the two kinds of extensions largely overlap in terms of frequency, and still overlap significantly when effort is taken into account. In this section, we seek to quantify the difference.

Given the large size of our sample, a statistical test such as the non-parametric Mann-Whitney U-test would almost certainly find a difference in the values of the distributions, and being in favor of data extensions. However, such a test would not tell us anything about the magnitude of the difference. As such, we measure the effect size of the differences in the distributions.

The most well-known effect-size metric is Cohen's d; however, it is not robust to departures from normality. As such, we opted for a non-parametric effect size, Vargha and Delaney's $\hat{A}_{12}$ [25]. This effect size measure was recommended by Arcuri and Briand in the case of algorithms whose performance follow geometric distributions which strongly depart from normality [1]. $\hat{A}_{12}$ ranges from 0 to 1, and measures the probability that a value taken at random from the first sample is higher than a value taken at random from the second sample.

In the case of unweighted frequencies of both kinds of extensions, we obtain an $\hat{A}_{12}$ value of 0.5554 in favor of data extensions, *i.e.* there is a 55% probability that a randomly chosen frequency of data extension is higher than a randomly chosen frequency of operation extension. This is very close to 50%, where the effect would be null. Since Cohen's d has well-accepted thresholds for effect sizes, we computed an estimate of the equivalent Cohen's d for this value. Our estimation of Cohen's d gives us 0.03, an effect that is considered as *trivial*, barely worth mentioning[6].

If we weight the measurements by effort, the picture is somewhat different. The advantage towards data extensions increases, with $\hat{A}_{12}$ being 0.6197:

[6] Cohen's d varies from -1 to 1; the commonly accepted thresholds for effect size are 0.2 (small), 0.5 (medium), and 0.8 (strong). Negative values of d indicate an effect in the opposite direction, and have identical thresholds.

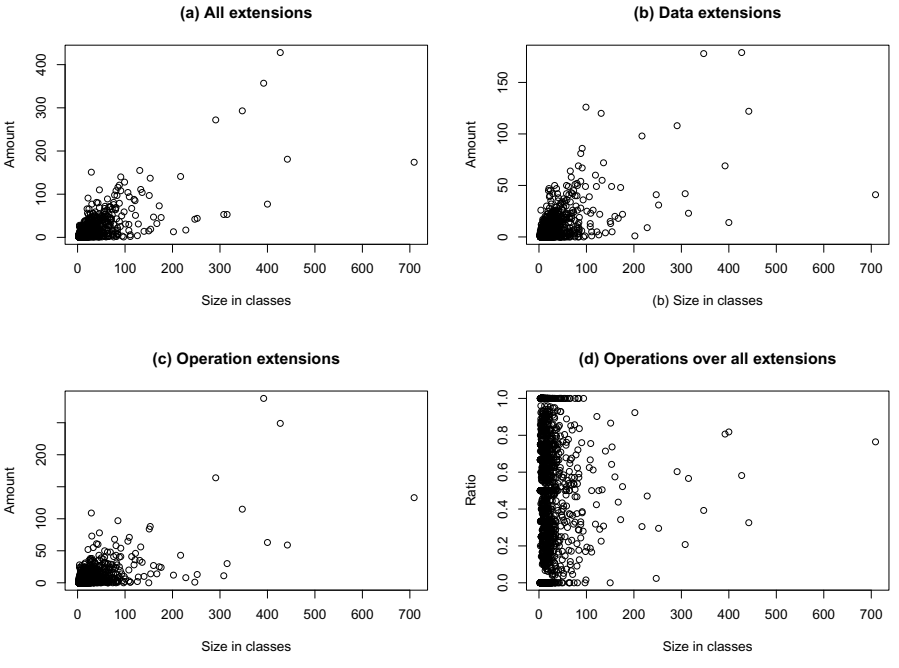


Fig. 7. Effect of size of hierarchies on kinds of extensions

A randomly-chosen weighted data extension count has a 62% probability of being larger than a randomly picked weighted operation extension count. If this higher probability seems reassuring, we do not know how to interpret that value. We again computed an estimate of the equivalent Cohen's d for this probability; we obtained a value of 0.25, which gives us a *small* effect. In other words, if data extensions are more common (barely), and involve more effort (somewhat), a large part of the extensions still are done by adding operations. For the object-oriented paradigm to be most suited for most extensions, we would have expected a much larger advantage in favor of data extensions, with at least a *medium*, if not a *large* effect size.

We quantified the effect size at the level of projects, where we obtained nearly identical results ($\hat{A}_{12}$: 0.5514 (unweighted) and 0.6307 (weighted); estimate of d: 0.05 and 0.25). These findings show that in practice, both kinds of extensions are needed in object-oriented programs; as such, adequate means to express both kinds of extensions are required in order to assist developers.

5.3 Relationship with Size of Hierarchies

We now look at how the size of hierarchies affects the number of extensions and their kinds. The scatterplots in Figure 7 show the relationship between size of hierarchies and: all extensions (a); data extensions (b); operation extensions (c); and ratio of operation extensions over all extensions (d).

To quantify the relationships, we measure the Spearman correlation between the number of extensions and the size of the hierarchies. Correlation ranges from 1 (perfectly correlated), to -1 (perfect inverse correlation), with 0 being uncorrelated. Spearman's ρ is a rank-based, non-parametric correlation, and as such it is less sensitive to outliers than alternatives (*e.g.* Pearson's product-moment correlation). Commonly-used thresholds for correlation are: 0.1 (small), 0.3 (medium), and (0.5) strong. We also report the statistical significance of the correlations we encounter, using the common threshold of $p < 0.05$ for significance. All the correlations below are highly statistically significant: in all cases, $p \lll 0.01$.

We start with both kinds of extensions taken together (Figure 7a). We see an upward trend (large hierarchies have more extensions) and find a strong correlation ($\rho = 0.67$). This corroborates our findings in Section 4.2, where we found that 80% of the hierarchies with five or more classes had extensions.

Figure 7b shows the relation between the size of hierarchies and the number of data extensions. We see an upward trend as well, giving us the impression that overall larger-sized hierarchies have more data extensions. The Spearman correlation yields a value of $\rho = 0.48$, which qualifies for a *medium* correlation.

The same situation holds with respect to the relationship between operation extensions and size, as shown in Figure 7c. Surprisingly, we observe a higher correlation between size and number of operation extensions, passing the *strong* threshold, with $\rho = 0.55$. If we weight the observations, we see an increase in the correlation for operation extensions ($\rho = 0.59$), and a decrease in the correlation for data extensions ($\rho = 0.42$). We have seen previously that both kinds of extensions are prevalent, with a small advantage for data extensions; here, operation extensions tend to increase more with the size of the hierarchies.

Having observed that operation extensions seem to "take the edge" in large hierarchies, we investigated if this behavior extends to the proportion of extensions. We computed the ratio of operation extensions over all extensions, and investigated its relationship to size. However, as Figure 7d shows, we found no visible relationship: hierarchies are nearly evenly spread across the ratio spectrum. Since the overall difference in correlation was not very large, the relationship practically disappears when the ratio is taken into account. Clearly, there are other factors at play also influencing the relationship between the two variables, as we see next.

In the interest of completeness, we mention that relationships taken at the project level exhibit a similar behavior, having significant, medium-to-strong correlations in the first three cases (a, b, and c).

5.4 Executive Summary

Analyzing the frequency and the effort invested in each kind of extension, we see overall that data extensions are slightly more frequent than operation extensions. However, this difference is very small: operation extensions are mostly as frequent as data extensions, and only somewhat smaller. If the extension mechanisms of object-oriented programming was adequate in most cases, the proportion of data extensions would be much larger.

To make matters worse, while both kinds of extensions are unsurprisingly correlated with the size of hierarchies, we find that operation extensions are actually slightly more correlated with size. Large hierarchies seem to necessitate more operation extensions.

6 Extensions and Evolution

In the previous section, we have seen that even if both kinds of extensions are correlated with the size of hierarchies, the ratio of operation extensions over both extensions was not obviously correlated with size. However, there may be other factors influencing this ratio. In particular, Lehman's laws of software evolution [13] say that software systems tend to decay over time, if no effort is undertaken to prevent that. Thus it seems reasonable to think that over time, unanticipated design decisions lead to more extensions that do not fit the class hierarchy, and as such need to be done via operation extensions. Hence, we analyze the proportion of operation extensions out of all extensions over time.

6.1 Introducing Periods

To answer this question we split the evolution of class hierarchies in periods. We gather all the commits affecting a candidate hierarchy, sort them according to time, and split the resulting list in 50 slices, each representing one period of the evolution. If a hierarchy was changed less than 50 times, we distribute the changes across the periods as close to being equidistant as possible. Since there is considerable variation in the number of changes between hierarchies, this ensures a uniform distribution of the changes over the 50 periods[7].

We then aggregate all the changes of all the hierarchies that belong to the same period. For each of these sets of changes, we sum the number of operation and data extensions, and compute the ratio of data extensions over all extensions, resulting in a proportion between 0 and 1 for all periods.

We also investigate the phenomenon at the level of projects; there, the only difference is that we gather all the changes related to a project before splitting the history in 50 periods. If a hierarchy is added later in a project, its changes will be distributed across the later periods of the project evolution only.

6.2 Evolution of the Ratio of Operation Extensions

Figure 8 plots the evolution of the proportion of operation extensions among all extensions over time, considering both hierarchies (a) and projects (b). To highlight the overall trend, a smoothed fitted curve is added to the scatterplots.

[7] We contemplated splitting the sets of changes in equal time periods, instead of equal number of commits per period. However, determining the time periods involves computing the time interval based on the first and the last change of the hierarchies. This introduces a bias in the earlier and later periods (more changes are found in the very first and very last periods), hence we discarded that idea.

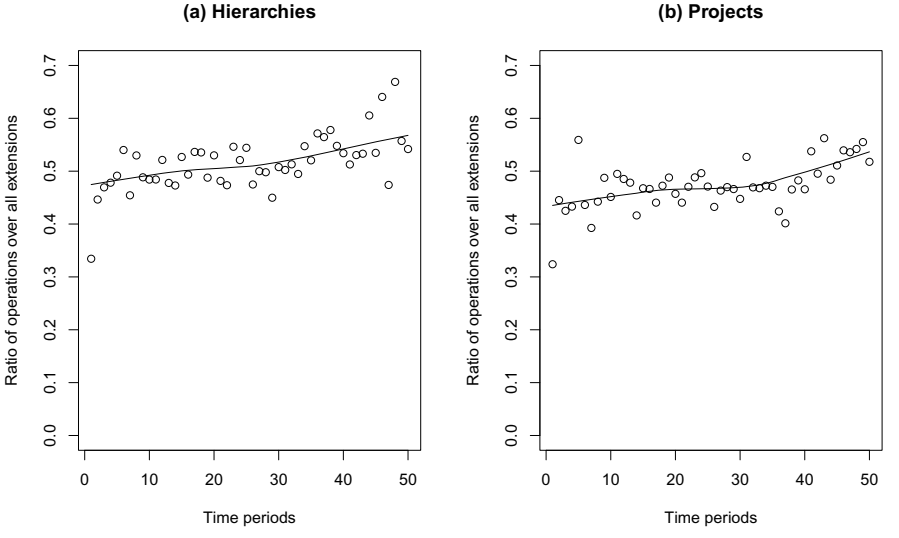


Fig. 8. Proportion of operation extensions among all extensions over time. (a) hierarchies; (b) projects.

In both cases, there is clearly an increase of the ratio of operation extensions over all extensions over time. The effect is more pronounced when hierarchies are considered on their own, which is not surprising: a possible reason being that new hierarchies may be added to projects later on. These hierarchies will then be "younger" and for a while offset the upward trend. It is interesting that the smoothed curve on the project scatterplot rises more sharply in the last periods: a possible explanation is that by then, the "young" hierarchies have begun to also become older, and seen their ratio increase, in turn impacting the project.

After a visual check, we quantify the relationship. The Spearman correlation indicates for both cases a significant relationship, which also confirms the visual impression that the effect is more pronounced for hierarchies in isolation than it is for projects. We find a Spearman correlation of $\rho = 0.60$ ($p \lll 0.01$) for hierarchies, and of $\rho = 0.51$ ($p \lll 0.01$) for projects.

If we take weighting into account (not shown in the figure), the relationship—unsurprisingly—drops. It however stays significant. The correlation of the weighted ratio with time for hierarchies is $\rho = 0.38$ ($p = 0.007$), and for projects, $\rho = 0.33$ ($p = 0.018$, less than the usual 0.05 threshold).

Of course, these correlations are not very strong; nor should we expect them to be. There are many more factors, beyond mere time passing, that could explain why a given hierarchy may need more of a certain kind of extension than others.

6.3 Executive Summary

If we consider the higher ratio of operation extensions as a sign of decay of object-oriented software, these results certainly confirm Lehman's observations

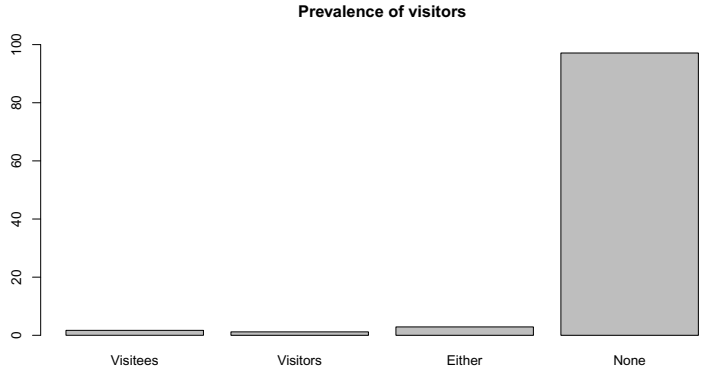


Fig. 9. Proportion of visitors, visited hierarchies, across all hierarchies

that software systems decay over time. We have found a moderate, yet significant, relationship between the ratio of operation extensions over all extensions and the age (as changes per periods), for both hierarchies and projects.

In our overall analysis, this adds evidence towards the emerging trend that more complex hierarchies (*i.e.* larger, older, etc.) are more confronted with extensions that do not fit the paradigm than other ones. Further, they seem to require proportionally more operation extensions than data extensions. These results cement the relevance of supporting both kinds of extensions adequately, as the most problematic hierarchies are the ones that need solutions the most.

7 Is the Visitor Pattern a Suitable Solution?

The well-known solution to operation extensions in object-oriented software is the Visitor pattern [7], as briefly described in Section 2.1. Is the Visitor pattern enough? We first analyze the prevalence of visitors in our data set, and then look at how both visitor hierarchies and the hierarchies they visit are themselves subject to operation extensions.

7.1 Prevalence of the Visitor Pattern

Figure 9 shows the results of our categorization of the hierarchies according to our visitor detection algorithm (Section 3.2). One can clearly see that a minority of classes are involved as either visitors or visitees. Out of the 2,879 hierarchies that experienced at least an extension (shown in the figure), 34 are visitors, and 49 are visitees, corresponding to a total of 2.88% of these hierarchies. In all the 10,271 hierarchies, we find 57 visitor hierarchies, and 62 visited hierarchies, for an even smaller proportion of 1.16%[8].

[8] The discrepancy in number is because there may not be a 1-to-1 mapping between visitors and visited hierarchies.

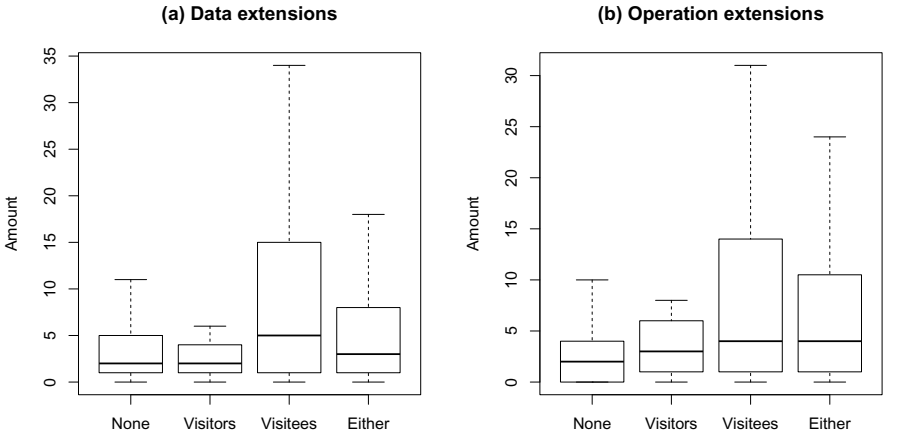


Fig. 10. Distribution of data and operation extensions by role in the Visitor pattern

All in all, usages of the Visitor pattern are few and far between. If it alleviates the issue of dealing with operation extensions, it cannot do so on a large scale, either because it cannot cover all cases, because few programmers have knowledge of the pattern (which, considering the popularity of design patterns, seems somewhat unlikely), or because the adoption cost of the pattern is judged too high. We also notice that the proportion of visitor and visited classes that experience extensions (83 out of 119, or 69.7%) is much higher than the proportion of hierarchies overall (2,879 out of 10,271, or 28%). This seems to indicate that classes involved in the visitor pattern are extended more than other classes. This warrants further investigation.

7.2 How Are Visitors and Visitees Extended?

Considering the documented drawbacks of the Visitor pattern (adding a new class in the visited hierarchy impacts all the visitors), we would expect the uses of the Visitor pattern to follow the Gang of Four's recommendations, and be applied to *stable* visited hierarchies only [7]. This means that visited hierarchies should feature less data extensions, and the corresponding visitor hierarchies should undergo less operation extensions.

Figure 10 shows the distribution of extension metrics, contrasting normal hierarchies, visitor hierarchies, visited hierarchies, and the last two kinds of hierarchies taken together. What we see contradicts our intuition. First, in Figure 10a, visitors seem to exhibit the same number or less data extensions than normal hierarchies, and visited hierarchies seem to feature considerably more! In Figure 10b, we see that visitors seem to feature slightly more operation extensions, and visitees considerably more.

There is, however, an important confounding factor: size. If a hierarchy has many data extensions, it is by definition large. Examining the data, we found that, indeed, an overwhelmingly large proportion of visited hierarchies are large. Thus we account for size in our statistical tests to assess whether the observed effect is significant.

Since there is no statistical test to determine the impact of a confounding factor [16], we employ an alternative strategy. We test for the statistical significance of differences in the number of data and operation extensions, for both visitor and visited hierarchies compared to normal hierarchies, using the Mann-Whitney U-test (the non-parametric equivalent to Student's t-test), under the null hypothesis $H0$ that there is no difference between the quantities of data and operation extensions. In case we find a significant difference and hence reject $H0$, we control for the size factor by doing a subsequent Mann-Whitney test, but this time comparing only the hierarchies with more than five classes.

After doing this procedure, the only significant differences below the $p = 0.05$ threshold were the number of data and operation extensions of visited hierarchies compared to normal hierarchies. However, most visited hierarchies actually have more than ten classes (44 out of 49). Repeating the same procedure but with a threshold of ten classes, both relationships lose their significance[9].

All in all, we can safely assume that any relationship between role (*i.e.* visitor, visitee) in the Visitor pattern and number of extensions is primarily due to the confounding factor of size. This makes classes involved in the Visitor pattern no worse, but also no better, than regular classes, for both roles and both metrics. Overall this suggests that the GoF advice of using the Visitor pattern on stable hierarchies may not be followed in practice. We have observed several examples of operation extensions in visitors that were performed to retrofit the visitors to data extensions in the visited hierarchies.

7.3 Executive Summary

We find that the Visitor pattern is not used very often in our dataset. Further, visitor and visited hierarchies seem to feature the same rate of extensions as other hierarchies (when accounting for size). We can conclude that the Visitor pattern is a viable solution only for a subset of all the extension cases. In addition, we noticed that visited hierarchies still suffer from operation extensions, which should normally be handled in the visitors. Finally, the results show that the GoF advice—the Visitor pattern should be applied only to stable hierarchies—is hardly followed in practice. This differs from Aversano's study, which found that visited hierarchies were stable, albeit on three systems only [2].

8 Threats to Validity

In this section we report on the threats to validity of our study. We distinguish between (i) construct validity, that is, threats due to how we operationalized

[9] The p-value of 0.06 for operation extensions is close to significance; however, raising the number of classes further eliminates this tenuous relationship.

the measures, (ii) internal validity, that is, threats affecting the measured cause-effect relationship, and (iii) external validity, which refers to threats concerning the generalization of the experiment results.

8.1 Construct Validity

By weighting each data extension with the number of methods added along with the new class, we might not correctly represent the severity of a data extension. For instance, after the initial addition of the class in a particular commit, the class might be extended with more methods in subsequent commits, methods that should also be considered when weighting this data extension.

The various thresholds we impose during data analysis (*e.g.* only class hierarchies with more than two classes and that have been changed more than five times are studied), have an influence on how many data and operation extensions we measure. However, we carefully selected these thresholds empirically, that is, by experimenting with different threshold values. The currently selected thresholds are most reasonable given the analyzed data. In the case of the threshold for large commits (addition of more than 50 entities in a commit), we observed that some genuine operation extensions were actually above that threshold; for instance, a polymorphic method was added on 61 classes of the same hierarchy in a single commit.

Since we do not analyze the source code inside methods, we do not account for methods that perform an explicit dispatch based on the type of an object in a functional design manner (*e.g.* `anObject isFoo ifTrue: [...] ...`). These methods are in fact operation extensions in disguise, for which the developer did not adopt the object-oriented paradigm in order to avoid having to add methods in scattered places. As such, we may under-estimate the amount of operation extensions that are performed.

Another source of underestimation of operation extensions is that we do not consider the *class extension* mechanism of Smalltalk. Class extensions are methods added to existing classes in one project by another project. For instance, the class `Object` in the kernel of Pharo Smalltalk has several dozens methods defined by other projects. These are excluded from our analysis, and may reflect potential operation extensions. We counted the number of methods defined as class extensions, and found that they represented 3.79% of all methods (2,732,618 out of 72,028,070 method definitions across all versions). As such, they are unlikely to influence our results.

We only study additions of methods and classes, not their modifications. If we considered modifications as well, we may find a higher proportion of changes related to data and operation extensions, for instance because such extensions tend to trigger more modifications than other additions.

8.2 Internal Validity

Squeaksource contains a considerable amount of code duplication, since projects are stored several times in the repository, for instance once as an individual

project and once embedded in another project. In a recent study, we found that 10-15% of the code in Squeaksource is duplicated [21]. This aligns with the code duplication rate found in the literature [11,14]. The effect of the presence of code duplication on the results of our study is hard to predict. We assume that duplicated projects do not stand out regarding data or operation extensions and hence expect the effect of code duplication to be minimal.

If the same method is added to two unrelated siblings, we count this as an operation extension, even if all other classes in the hierarchy do not either define or inherit the method. Such a case may either be an incomplete operation extension, two unrelated single-method extensions, a bug, or an incremental step towards a consistent extension. In a dynamically-typed language, it is hard to tell whether this scenario corresponds to an operation extension or not, unless we rely on human judgment. This is because object interfaces are totally implicit in such languages. In a statically-typed language, object interfaces are explicit and the type system ensures that an extension of the interface is consistently implemented.

The detection of renames of root classes in a hierarchy is not perfect and might not detect some renames. In such a case we end up with having an old, obsolete hierarchy in our dataset to which we cannot relate any subsequent changes and thus not detect operation or data extensions affecting such a hierarchy. We however expect such cases to be rare and could not find a single false-negative case while overviewing most of the very large hierarchies in Squeaksource.

The visitor detection heuristic we implemented is also not perfect. However, we validated each identified visitor manually and did not find any false-positives, thus the detection algorithm yields a precision of 100%. The recall is not assessable though, our algorithm might not detect all visitors, thus we possibly underestimate the presence of visitors and visited hierarchies. Since we search for variations in terminology (*e.g.* `accept` and `visit` for visitor methods), we expect the recall to be fairly high.

We took dispositions against the ecological fallacy [17]—incorrectly assuming that observations holding at a level of abstraction holds at another level—by systematically verifying that findings we found at the level of class hierarchies also applied at the level of projects, when it was pertinent to do so.

8.3 External Validity

The generalization of our study is dependent on how representative the analyzed projects are for object-oriented software projects in general. As Squeaksource is a very large repository containing more than 2,500 projects to which more than 2,300 developers contributed, we expect that very different programming styles and flavors have been applied in these projects, making the analyzed projects well representative of object-oriented software.

A possible bias is that our sample of project contains only open-source software systems. Practices in the industry may differ and limit the generalization of our results. However, access to a large sample of closed-source software systems is notoriously difficult.

Smalltalk is a dynamically-typed programming language. In a statically-typed language, data and operation extensions might be employed differently, following different rules and patterns. It is very hard to assess whether one or both type of extensions are more or less frequent in a statically-typed languages than in its dynamic pendant. Also, we cannot claim that the results we found for Smalltalk also hold for other dynamically-typed object-oriented languages, although we expect to find similar patterns. It would be very interesting to replicate our study for *e.g.* Java and Ruby, to assess the use of data and operation extensions in other object-oriented languages.

Smalltalk is an object-oriented language. The extensibility challenge we studied is a general problem that occurs with other abstraction mechanisms as well. We cannot claim that the results related to the kinds of extensions that occur in Smalltalk projects also apply to other mechanisms. Studying programs written in languages with different mechanisms (*e.g.* ML, Haskell), including combinations of objects and others (*e.g.* Scala, Racket), would be extremely interesting to shed more light on this topic.

9 Conclusions

Reconciling the two kinds of extensions to data types has been a subject of interest for years, if not decades; we assessed the prevalence of this challenge with a large-scale empirical study. Our empirical study of the Squeaksource ecosystem analyzed more than half a billion lines of code, distributed over 2,505 projects and 111,071 commits. Thousands of contributors performed these commits over the course of 8 years.

We found the following:

1. Extensions do occur: one out of eight commits introduces an operation or a data extension; large projects and large hierarchies are more prone to extensions. More than half of the large class hierarchies have to be extended with new operations.
2. Both kinds of extensions happen with roughly the same frequency. When effort is measured, data extensions take a small advantage. However, the margin is very small, so the data-extension friendly mechanism of objects needs supplementation for operation extensions.
3. Over time, projects and hierarchies tend to need more operation extensions, as the new extensions were not envisioned by the initial design. These larger, older hierarchies need better extensibility support all the more.
4. The Visitor pattern, the de-facto solution to modularly support operation extensions in object-oriented software, is not applied frequently. Furthermore, classes involved in the pattern still need operation extensions: in visited classes when the extensions do not fit well the Visitor pattern, and in visitor classes to react to data extensions in the visitees.

We see these findings as a call to the community to continue investigation on this topic, and, perhaps more crucially, to propose solutions to practitioners.

If the first can be done with novel languages, perhaps tool support is best to assist practitioners working on existing systems. For instance, IDEs could provide programmers with a way to switch between a data-centric view and an operation-centric view of the program. The seed of such tool support already exists in the venerable Smalltalk class browser, which is able to display all the implementors of a polymorphic method in a single, editable view. As for the extensibility problem stricto sensu, further studies are needed to see if our results also reflect black-box third-party extension scenarios.

Acknowledgments. We thank the ECOOP reviewers for their helpful comments. R. Robbes and É. Tanter are partially funded by FONDECYT Projects 11110463 and 1110051, respectively. D. Röthlisberger is funded by the Swiss National Science Foundation, SNF Project No. PBBEP2 135018.

References

1. Arcuri, A., Briand, L.C.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of ICSE 2011, pp. 1–10 (2011)
2. Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., Di Penta, M.: An empirical study on the evolution of design patterns. In: Proceedings of ESEC/SIGSOFT FSE 2007, pp. 385–394 (2007)
3. Baxter, G., Frean, M.R., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.D.: Understanding the shape of Java software. In: Proceedings of OOPSLA 2006, pp. 397–412 (2006)
4. Booch, G.: Object-Oriented Analysis and Design with Applications, 2nd edn. Addison-Wesley (1994)
5. Cook, W.R.: Object-Oriented Programming Versus Abstract Data Types. In: de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1990. LNCS, vol. 489, pp. 151–178. Springer, Heidelberg (1991)
6. Erlikh, L.: Leveraging legacy system dollars for e-business. IT Professional 2(3), 17–23 (2000)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley (October 1994)
8. Gîrba, T., Lanza, M., Ducasse, S.: Characterizing the evolution of class hierarchies. In: Proceedings of CSMR 2005, pp. 2–11 (2005)
9. Gorschek, T., Tempero, E.D., Angelis, L.: A large-scale empirical study of practitioners' use of object-oriented concepts. In: Proceedings of ICSE 2010, pp. 115–124 (2010)
10. Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Q., Ghezzi, C.: An empirical investigation into a large-scale java open source code repository. In: Proceedings of ESEM 2010, pp. 11:1–11:10 (2010)
11. Kapser, C.J., Godfrey, M.W.: Supporting the analysis of clones in software systems: A case study. Journal of Software Maintenance and Evolution: Research and Practice 18 (2006)
12. Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing Object-Oriented and Function Design to Promote Reuse. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 91–113. Springer, Heidelberg (1998)

13. Lehman, M., Belady, L.: Program Evolution: Processes of Software Change. London Academic Press, London (1985)
14. Mayrand, J., Leblanc, C., Merlo, E.M.: Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings of Software Maintenance, pp. 244–253 (November 1996)
15. Oliveira, B.C.d.S.: Modular Visitor Components: A Practical Solution to the Expression Families Problem. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 269–293. Springer, Heidelberg (2009)
16. Pearl, J.: Why there is no statistical test for confounding, why many think there is, and why they are almost right. Technical report, Department of Statistics, University of California, Los Angeles (1998)
17. Posnett, D., Filkov, V., Devanbu, P.: Ecological inference in empirical software engineering. In: Proceedings of ASE 2011, pp. 362–371 (2011)
18. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to data abstraction. In: New Advances in Algorithmic Languages, pp. 157–168 (1975)
19. Robbes, R., Lungu, M.: A study of ripple effects in software ecosystems. In: Proceedings of ICSE 2011, NIER Track, Honolulu, Hawaii, USA, pp. 904–907. ACM Press (May 2011)
20. Van Rysselberghe, F., Demeyer, S.: Studying versioning information to understand inheritance hierarchy changes. In: Proceedings of MSR 2007, p. 16 (2007)
21. Schwarz, N., Lungu, M., Robbes, R.: On how often code is cloned across repositories. In: Proceedings of ICSE 2012, NIER Track (2012)
22. Shalloway, A., Trott, J.R.: Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd edn. Addison-Wesley (2004)
23. Tempero, E., Noble, J., Melton, H.: How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 667–691. Springer, Heidelberg (2008)
24. Torgersen, M.: The Expression Problem Revisited: Four New Solutions Using Generics. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
25. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)
26. Wadler, P.: The expression problem. Mail to the java-genericity mailing list (1998)
27. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: Proceedings of FOOL 2005, Long Beach, USA (January 2005)
28. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Transactions on Software Engineering 31(6), 429–445 (2005)

PQL: A Purely-Declarative Java Extension for Parallel Programming

Christoph Reichenbach[1], Yannis Smaragdakis[1,2], and Neil Immerman[1]

[1] University of Massachusetts, Amherst
[2] University of Athens, Greece
{creichen,yannis,immerman}@cs.umass.edu

Abstract. The popularization of parallelism is arguably the most fundamental computing challenge for years to come. We present an approach where parallel programming takes place in a restricted (sub-Turing-complete), logic-based declarative language, embedded in Java. Our logic-based language, PQL, can express the parallel elements of a computing task, while regular Java code captures sequential elements. This approach offers a key property: the purely declarative nature of our language allows for aggressive optimization, in much the same way that relational queries are optimized by a database engine. At the same time, declarative queries can operate on plain Java data, extending patterns such as map-reduce to arbitrary levels of nesting and composition complexity.

We have implemented PQL as extension to a Java compiler and showcase its expressiveness as well as its scalability compared to competitive techniques for similar tasks (Java + relational queries, in-memory Hadoop, etc.).

1 Introduction

Parallelism is here to stay. Parallel hardware has already transitioned from niche architectures to mainstream computing. Power and latency trends (of electronics, as well as of other foreseeable physical processes) dictate that the computer industry shift permanently to parallel processing, instead of trying to improve on traditional single-core sequential designs. Programming parallel computers, however, is a formidable challenge. Various forms of parallel hardware have been around for decades, and generation after generation of programmers have been unable to utilize such hardware fully and easily through traditional programming models: Although there are well-known parallel algorithms, the base algorithmic thinking in computer science is sequential. Even worse, it is not the case that we can "start from scratch" and disregard sequential computation. A pure parallel future is unlikely. Fast sequential processing is the greatest advantage that digital computers hold over massively parallel natural computers, such as the human and animal brain. Thus, it seems inevitable that we are heading towards a future where we will need to program both sequential and parallel algorithms in a unified manner.

Our work advances parallel programmability through a unified sequential-parallel programming model, reified in a language design we call PQL/Java. We use a general-purpose programming language (Java) as the substrate and extend it with PQL: a declarative, logic-based sublanguage (based on first-order logic operators). Any program expressible in PQL will be automatically parallelized. In fact, the expressiveness of PQL is explicitly limited so as to allow efficient parallelization.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 53–78, 2012.

For a preliminary example of PQL, consider a simple program fragment:

```
int[] arr = query (Array[x] == y): range(1, 1000).contains(x) && y == x*x;
```

The above Java declaration uses a PQL expression to initialize an **int** array variable. The PQL expression starts by stating the form of the result: it will be an array (PQL keyword Array is one of three possible at this position, the others being Map and Set) mapping x to y. The body of the query specifies that y is the square of x, for x between 1 and 1000. ("range" is a library method that produces sets.) PQL will parallelize the evaluation, if deemed profitable, and split it among the available processors. (We give this and several other very short examples only language illustration: parallelization will not help for such simple expressions and small data amounts.)

The distinguishing feature of PQL is that it is transparently integrated in Java yet fully declarative, without any order dependencies between clauses. The New Oxford Dictionary of English defines "declarative" in the context of Computing as "denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed". In other words, a program or language is declarative when it specifies *what* needs to be computed but not *how*. The "how" can be highly variable and the language implementation has a lot of choice in this decision.

To see what declarativeness means in our context, consider what is perhaps the closest conceptual relative of PQL: the .Net PLINQ facility for parallel queries [9]. PLINQ is also a parallel query language with explicit syntactic support (inside .Net).[1] PLINQ and PQL differ in a myriad of language design choices—e.g., logic-based sentences (forall/exists clauses) vs. relational queries (select–from–where clauses). But the deepest difference is that PQL is fully declarative, thus allowing far more optimization and transformation of the parallel query code, but also limiting what can be expressed in a query. Consider a query that combines two data structures: a set, premiumcustomers, and a map, cust2orders, returning a new data structure that maps every customer in the set to their high-value orders (e.g., above a value of 1000). Both PQL and PLINQ can easily express such queries. In the case of PQL, we have:

```
Map m = query (Map.get(cust) == order):
        premiumcustomers.contains(cust) &&
        cust2orders.get(cust) == order && order.amount > 1000;
```

Similarly, in PLINQ one might write (adapting to C# idioms and structures):

```
var m = from cust in premiumcustomers.AsParallel()
        from order in cust2orders[cust].AsParallel()
        where order.amount > 1000
        select new { cust, order };
```

The difference is that the implementation of the PLINQ query has fewer degrees of freedom than the implementation of the PQL query. The programmer needs to specify which traversals are done in parallel. Also, the traversals are pre-ordained: the system will iterate over all elements in set premiumcustomers and then over all elements in each set of orders for the given customer. In contrast, the PQL query offers no guarantee or

[1] Strictly speaking, the syntax extensions to VB and C# are for LINQ, the sequential querying interface, and PLINQ requires no extra support.

even indication of how parallelism is applied or the order of traversal. The PQL implementation is free to reorder the query clauses in many ways, since all PQL expressions are guaranteed side-effect-free. (PQL queries can contain arbitrary Java expressions that refer to Java program variables, but not to PQL "variables", i.e., can be evaluated just once for the entire query.) The system may decide to iterate over all elements of the cust2orders map first, or over all elements of the premiumcustomers set, or even over all objects of type Customer that exist on the heap, even if they are not guaranteed to be members of premiumcustomers. The latter will not be cost-optimal for this query, but either of the former two traversals may be, depending on the sizes of the data structures. Similarly the implementation of the query may choose to partition (and parallelize the traversal of) either the premiumcustomers set, or the cust2orders map, or even the traversal of all heap objects.

PQL can express many parallel tasks, but *its main strength is for generalized, arbitrarily nested map-reduce-like relation manipulations*. Indeed, the language is explicitly designed to combine logical queries and reduction operators. The query language design targets a specific level of expressiveness, in order to enable parallelization. The inspiration comes from complexity theory. "Descriptive complexity" [11] is the sub-area of complexity theory that matches logical languages with complexity classes. In terms of descriptive complexity, first-order logic over finite structures is a language that can express exactly the problems that can be optimally parallelized, i.e., solved in constant time by a CRAM (Concurrent Random Access Machine)—a theoretical abstraction of a parallel machine—using a polynomial number of processors. Of course, this highly theoretical view ignores important practical overheads and constraints (e.g., when finding the minimum of n elements, we do not want to perform n^2 comparisons in practice, even if these are in parallel). Still, the theoretical expressiveness class serves as a good guideline for our design, and we can classify PQL precisely as a *first-order* query language.

Generally, the main contributions of our work are as follows:

- We present a new approach to parallel programming, consisting of an embedding of a fully declarative query language inside a general-purpose language.
- We discuss the embodiment of our approach in a specific language setting and detail its essential features for expressiveness and optimization. Although one can discern high-level similarities of our PQL/Java language with others (e.g., database query languages embedded inside general purpose languages), our need for tight integration of the two language models creates unique demands and opportunities at both the language design and implementation level.
- We present performance measurements of PQL/Java for sample tasks to showcase its areas of strength and implementation scalability. The results validate the ease with which simple declarative tasks can exploit parallelism, reaching the performance of manually optimized code.

2 Language Illustration

We begin with a description of the PQL/Java language, with several examples interspersed for illustration.[2]

[2] Language specification and implementation are available at http://creichen.net/pql

2.1 Language Constructs Overview

At the high level, PQL is primarily a first-order query language. This means that it can be viewed as a first-order logic, with the usual boolean connectives ("and", "or", etc.) and quantifiers ("forall", "exists"). As in every first-order language, the main functionality is defined as specialized predicates and functions that can be used in this logic. Additionally, PQL has an extra-logical component: it adds the ability to aggregate query results in more powerful ways than allowed by the logic ("reduce" them).

In more concrete terms, PQL defines the keywords **query**, **reduce**, **forall**, **exists**, and **over**. It further re-purposes existing Java constructs, including many operators (such as &&, ||) and some invocation-like expressions (such as set.contains(element)). In this idea it follows the Java Query Language JQL [16]. To integrate with Java, these constructs assume the meaning defined in this document in source files that contain the import statement:

import static edu.umass.pql.Query;

In the absence of this statement, the syntax and semantics of a PQL/Java program are identical to those of a regular Java program.

Syntactically, PQL/Java extends Java by allowing any Java expression (*JAVA-EXPR*) to be a query (*QUERY*). (We show the full syntax of PQL later, in Figure 1, but explain it here incrementally.) A query, in turn, follows the production:

$$\mathit{QUERY} ::= \langle \mathit{QUANT\text{-}EXPR} \rangle \mid \mathsf{id} \mid \langle \mathit{JAVA\text{-}EXPR} \rangle \mid \langle \mathit{QEXPR} \rangle$$

(We inherit from Java the usual non-terminals *JAVA-EXPR*, for Java expressions, *JAVA-TY*, for Java types, and id , for Java identifiers.) That is, a query may be a quantifier expression (*QUANT-EXPR*) that quantifies one or more *logical variables* (e.g., forall x, y : a[x] > b[y]), a single identifier that references such a logical variable (such as x or y in the above example), or one of two unquantified expressions: an arbitrary Java expression (which may contain side effects but cannot use logical variables—i.e., variables declared inside the PQL query) or a Q-Expression (*QEXPR*), which may use logical variables and sub-queries but no side effects. Since Java expressions may contain PQL queries, it is possible to nest multiple queries in the same expression, though these must not share variables.

Quantifier Expressions. Quantifier expressions take one of the following forms:

$$\begin{aligned} \mathit{QUANT\text{-}EXPR} ::= {} & \langle \mathit{QUANT} \rangle \ \langle \mathit{ID} \rangle \ \text{':'} \ \langle \mathit{QUERY} \rangle \\ & \mid \textbf{query} \ \text{'('} \ \langle \mathit{MATCH} \rangle \ \text{')'} \ \text{':'} \ \langle \mathit{QUERY} \rangle \\ & \mid \textbf{reduce} \ \text{'('} \ \mathsf{id} \ \text{')'} \ \langle \mathit{ID} \rangle \ [\textbf{over} \ \langle \mathit{ID\text{-}SEQ} \rangle \ ] : \ \langle \mathit{QUERY} \rangle \end{aligned}$$

The first form of quantification is universal or existential quantification: *QUANT* may be either forall or exists. Such an expression has boolean value, true or false. The second form of quantification, a *container query*, constructs maps, sets, or arrays. The third and final form is a general-purpose reduction operation.

Universal and Existential Quantifications. Universal or existential quantification extends over an identifier *ID*, which can explicitly declare a type:

$$\mathit{ID} ::= \mathsf{id} \mid \langle \mathit{JAVA\text{-}TY} \rangle \ \mathsf{id}$$

For now, consider an example of the second form:

```
forall int x : x == x
```

This tests whether all x that are of type **int** are equal to themselves. This particular test should always evaluate to **true**. Similarly,

```
exists int x : x*x == -1
```

will test whether there exists an integer x whose square is equal to -1; this test will evaluate to **false**. (Of course, the system has no way of knowing this fact statically, hence the query will be evaluated in parallel over all **int**s.)

We refer to the logical variable occurring in the *ID* construct as the *query variable*. If a Java type (*JAVA-TY*) is present, the query variable is *explicitly typed*, otherwise the range of values for the variable is inferred. These two cases behave differently. Informally, the difference is that for queries

```
/* A */ forall int x : rel[x] > 0
/* B */ forall x : rel[x] > 0
```

the compiler will infer the static type for case B, and also infer that it should only consider values for x that occur in the domain of rel, whereas for case A it will consider all 2^{32} possible **int** values for x, regardless of the size of rel. This topic is discussed in more detail in Section 2.3.

Container Queries. A container query has the syntax

query '(' ⟨*MATCH*⟩ ')' ':' ⟨*QUERY*⟩

where a *MATCH* is one of the following:

MATCH ::= 'Set' '.' 'contains' '(' ⟨*ID*⟩ ')'
| 'Map' '.' 'get' '(' ⟨*ID*⟩ ')' '==' ⟨*ID*⟩ [**default** ⟨*QUERY*⟩]
| 'Array' '[' ⟨*ID*⟩ ']' '==' ⟨*ID*⟩

The first of the above productions then constructs a set, as in the following example:

```
query (Set.contains(x)): x == 0
```

This would construct a set of integers containing precisely the number zero. (The use of Set.contains in the syntax is an allusion to the method by the same name in the Java standard API Set interface, and similarly for Map.get.)

The second construction builds a map:

```
query (Map.get(x) == y): range(1, 10).contains(x) && y == x*x
```

This would construct a map of all numbers from 1 to 10 to their squares. range(1, 10) here is a logical constant and a Java expression, denoting a set of all integers from 1 through 10. By contrast, y == x*x is a PQL subexpression: both x and y are logical variables. Note that the above does not provide mappings for numbers outside the range. For example, index 0 of the generated map will be **null**.

Maps may contain a default clause. For instance:

```
query(Map.get(x) == y default -1): range(1, 10).contains(x) && y==x*x
```

This would construct the same map as above, except that all numbers outside of the range 1 through 10 are mapped to −1.

Our third construct builds arrays. For example,

```
query (Array[x] == y): range(1, 10).contains(x) && y == x*x
```

is the same as our map construction without defaults, with one exception: the missing array index (0) is filled in with the default value for the relevant type (i.e., 0 for integers). Thus, this will construct an 11-element array containing $0, 1, \ldots, 81, 100$. Any attempt to define the array at a negative offset raises an exception.

Reductions. The last kind of quantifier expression in PQL is a reduction, which follows the syntax below:

reduce '(' id ')' ⟨*ID*⟩ [**over** ⟨*ID-SEQ*⟩] : ⟨*QUERY*⟩

(*ID-SEQ* denotes a comma-separated sequence of *ID*.) The first parameter to a reduction is always a reduction operator, such as the built-in sumInt, sumLong and sumDouble operators. Consider:

```
reduce (sumInt) x : myset.contains(x)
```

This will sum up all numbers contained in myset after coercing them to **int**. The type of a reduce expression is the type of the value being reduced over (e.g., **int** for a sum of Java primitive integers).

Sometimes reduction requires additional free variables. We obtain these using the keyword over:

```
reduce (sumDouble) x over y : set.contains(y) && x == 1.0 / y
```

This will sum up the inverses of all numbers contained in the container set.

In later sections we will see all built-in reductors as well as how to provide user-defined ones.

```
QUERY      ::= ⟨QUANT-EXPR⟩ | id | ⟨JAVA-EXPR⟩ | ⟨QEXPR⟩
QUANT-EXPR ::= ⟨QUANT⟩ ⟨ID⟩ ':' ⟨QUERY⟩
             | query '(' ⟨MATCH⟩ ')' ':' ⟨QUERY⟩
             | reduce '(' id ')' ⟨ID⟩ [over ⟨ID-SEQ⟩ ] : ⟨QUERY⟩
QUANT      ::= forall | exists
QEXPR      ::= '(' ⟨QUERY⟩ ')' | ⟨QUERY⟩ ⟨BINOP⟩ ⟨QUERY⟩
             | ⟨QUERY⟩ instanceof ⟨JAVA-TY⟩ | ⟨UNOP⟩ ⟨QUERY⟩
             | ⟨QUERY⟩ '?' ⟨QUERY⟩ ':' ⟨QUERY⟩ | ⟨QUERY⟩ '.' 'get' '(' ⟨QUERY⟩ ')'
             | ⟨QUERY⟩ '[' ⟨QUERY⟩ ']' | ⟨QUERY⟩ '.' 'contains' '(' ⟨QUERY⟩ ')'
             | ⟨QUERY⟩ '.' id | ⟨QUERY⟩ '.' length | ⟨QUERY⟩ '.' size '(' ')'
BINOP      ::= '||' | '&&' | '|' | '&' | '^' | '%' | '*' | '+' | '-' | '/' | '>' | '<'
             | '<=' | '>=' | '!=' | '==' | '<<' | '>>' | '>>>' | '=>'
UNOP       ::= '!' | '~' | '-'
MATCH      ::= 'Set' '.' 'contains' '(' ⟨ID⟩ ')'
             | 'Map' '.' 'get' '(' ⟨ID⟩ ')' '==' ⟨ID⟩ [default ⟨QUERY⟩ ]
             | 'Array' '[' ⟨ID⟩ ']' '==' ⟨CM⟩
ID         ::= ⟨ID⟩ | ⟨JAVA-TY⟩ id
ID-SEQ     ::= ⟨ID⟩ | ⟨ID-SEQ⟩ ',' ⟨ID⟩
```

Fig. 1. PQL/Java syntax

Q-Expressions. A Q-expression (non-terminal *QEXPR* in Figure 1) has essentially the same syntax as regular Java non-side-effecting expressions, with the exception of

method calls, which are not supported in general (though we do borrow method call syntax for a number of set and map operations). Q-expressions can freely refer to logical variables, and form the basis of parallel computations in PQL. All familiar Java operators have the same meaning inside a Q-Expression (including emulating the Java exception behavior). The only new operator is =>, denoting logical implication: 'a => b' is equivalent to '(!a) || b', with ! being logical negation.

There are some slight differences in how operators can be used with different types. Q-expressions of the form q.get(x) or q[x] are equivalent. Both denote a map or array lookup (depending on the static type of q) and evaluate to the element indexed by x. For example, myArray.get(3) for an integer array myArray would obtain the 4th element of the array. This operation raises the same exceptions as regular Java array accesses might raise. Q-expressions of the form q.f are projections that obtain the contents of field f. Field f must be accessible from the context in which the query originates according to the rules of reflective field access in Java (i.e., the field may be private and in a different class, but field access via the Java reflection API must be permitted). Finally, Q-expressions of the form q.length or q.size() are equivalent. Both evaluate to the size of an array, java.util.Collection, or java.util.Map, as determined by the static type of q.

2.2 Examples and Expressiveness

Before we discuss more advanced language features and semantics, we present examples of useful queries, to establish the usage model more firmly.

Consider the following prototypical map-reduce task [6]: identifying a three-character word in a set of strings. We here represent the strings as arrays of bytes (akin to the assumptions of Dean and Ghemawat [6]) and store them in an array of arrays, data_array. We compute the set of arrays that contain the string of interest ('0', '1', '2') with the following PQL query:

```
result = query(Set.contains(byte[] ba)):
           exists i: data_array[i] == ba
            && exists j: range(0, RECORD_SIZE − 3).contains(j)
              && ba[j] == ((byte)'0') && ba[j+1] == ((byte)'1') && ba[j+2] == ((byte)'2');
```

(Here, RECORD_SIZE is the number of bytes in all strings. We could equivalently use ba.length or ba.size(), which are evaluated at run-time, but we choose to keep the Dean and Ghemawat scenario of having static knowledge that we can exploit in the query.)

The PQL runtime automatically parallelizes this query, as it deems appropriate. For instance, the implementation may iterate sequentially over the contents of data_array in search of an appropriate component array ba, but then search each ba in parallel for a matching index, j. In later sections we use this query for illustration and describe how we translate it into our intermediate language and optimize it.

For a more complex example consider an adaptation of a common map-reduce motivating scenario [7]. An application keeps track of entities called Pages and Sites. A Page object uniquely maps to a Site (i.e., every page has a unique site, while a site "owns" pages) and Pages can refer to other Pages (i.e., every page is mapped to a set of other pages). This information is captured by regular Java maps and sets, namely two data structures "Map<Page,Site> page2site" and "Map<Page,Set<Page>> refersTo". Imagine that the programmer wants to implement the following functionality: for each page p,

count the number of sites that own pages that refer to p. This is captured by the following PQL query:

```
Map<Page,Integer> result =
  query(Map.get(p) == i):
    i == reduce (sumInt) one over Site s:
      one == 1 &&
      (exists Page pReferrer:
        page2site.get(pReferrer) == s && (refersTo.get(pReferrer)).contains(p));
```

In words, the program text says: compute a map from each page p to an integer i, so that i is the number of sites s (count one for each site) that own a page (and possibly more than one) that refers to p.

The query is automatically optimized, parallelized and executed by multiple processors. We can see some of the optimization reasoning in intuitive terms. The runtime system will first identify that there is a loop "exists pReferrer" over Page objects, another loop over Site objects (using variable s, over which we reduce), and implicitly a loop over variable p of type Page (which appears in the result). Thus, the query can certainly be evaluated by enumerating all n^3 combinations of values for p, s, and pReferrer. We can do better than that, however, since these values are related. The current pReferrer object is enough to index all relations used in the query (page2site and refersTo) and bind the values of p and s. To retrieve only relevant pReferrer objects, the system can partition the refersTo data and assign a portion to each processor. The results of each processor's computation will then be combined and finally reduced.

To see high-level optimization reasoning in more depth, consider another example over the same relations, page2site and refersTo. We would like to compute for every site the number of pages it owns that have outside references. This is accomplished by the following query:

```
Map<Page,Integer> result =
  query(Map.get(s) == i):
    i == reduce (sumInt) one over Page p:
      one == 1 && page2site.get(p) == s &&
      (exists Page pReferrer:
        page2site.get(pReferrer) != s && (refersTo.get(pReferrer)).contains(p));
```

For this example, an efficient evaluation will likely not start by traversing either instance of relation page2site. Examining an element of this relation does not lead to an efficient indexing of the other two relations involved in the query. (This is because the page2site map is efficient for retrieving sites given a page, but not vice versa.) Instead, a good evaluation order would start by enumerating relation refersTo and using the values of pReferrer and p that it obtains to index into the two instances of relation page2site. Furthermore, the expression "page2site.get(pReferrer) != s" does not really bind variable s: examining tuples of page2site for a given pReferrer tells us what s is *not*. Therefore, the optimal join order (for reasonable assumptions of relation sizes) is to start with refersTo, proceed to the first instance of page2site and then to the second. We see that an automatic optimizer is highly desirable even for small queries. For larger queries, it quickly becomes invaluable and relieves the programmer of the obligation to think about evaluation details, instead concentrating on the specification of the desired task.

In general, the optimizer needs to find the evaluation strategy (i.e., program transformations and join order) that binds all variables in the result tuple with the minimum lookup cost and space requirements for intermediate results. Any iteration over the elements of a type or a relation whose size exceeds a pre-defined threshold can be parallelized for efficiency: all that is required is a partitioning of the relation and assignment of each partition to the appropriate processor. A welcome property is that the problem of parallelization becomes *easier* the larger the size of the data involved in a query. Ideally, a single relation is partitioned and the rest of the evaluation logic (i.e., joining the rest of the logical conditions) is executed sequentially by the processor assigned to the partition.

Overall, PQL is quite expressive and allows arbitrary combinations of queries. Effectively, every query that one can express in relational algebra or SQL can also be expressed in PQL (since PQL includes full first-order logic plus aggregation operators, in the form of built-in reducers), although the difference in the structure of the query can be significant. In theoretical terms, this is exactly the class of queries that can be parallelized optimally, i.e., executed in constant time if the number of processors is large (but still polynomial relative to the input size) [11, Theorem 14.9, 5.27].[3] However, one should be careful in interpreting complexity theory results in a practical setting: although an algorithm expressed in such a query language can be parallelized optimally, this is useless if expressing the algorithm in the language greatly grows the cumulative work to be done by all processors together, e.g., from $\Theta(n)$ to $\Theta(n^2)$.

2.3 Beyond Basics

To complete our informal description of the language, we next discuss some important design issues: types, exceptions, our notion of equality, and user-specified reductors.

Types. PQL type checking generally imitates Java, with some exceptions. Types are implicit, unless annotated explicitly. As we have seen, logical variables can either be declared 'with type' (as in forall **int** i : ...) or 'without type' (as in forall i : ...). These declarations specify different semantics. The explicitly typed variant has the obvious semantics, for example, forall **int** i : i == i will loop over all 2^{32} integer values and check that they are equal to themselves. At runtime, explicitly typed variables conceptually iterate over all *viable* values of their type τ, where *viable* is defined as follows:

- If τ is an enumeration, the viable values of τ are all members of the enumeration, as per java.util.EnumSet.allOf.
- If τ is an ordinal type such as **int** or **boolean**, the viable values of τ are all possible values for τ permitted by the Java programming language.
- If τ is a floating-point type, i.e., **float** or **double**, then the viable values for τ are all the values of that type that exist in live objects on the Java heap.[4]

[3] Strictly speaking, this bound makes unrealistic assumptions with respect to the complexity of merging results (especially for reduction operations). Still, the practical approximation of such theoretical models typically incurs a $log(n)$ slowdown factor, where n is the input size, which is perfectly acceptable as a bound for parallelization purposes.

[4] Support for iterating over floating point and reference values requires a runtime with either heap traversal functionality (through a VM extension) or load-time code rewriting. We show that finding objects by type on the heap can be efficient in our earlier DeAL system [13], though our current PQL implementation does not yet replicate this feature.

- Otherwise, τ is a reference type, and the viable values for τ are all the objects of that type that exist in live objects on the Java heap.

By contrast, a logical variable without explicit type is subject to *domain inference*. This means that the variable's type and bounds are inferred from the body of the quantification. In terms of static types, we infer the type of a query variable as the least upper bound of all the types it can assume, eagerly defaulting to java.lang.Object at the least precise. In terms of runtime values, domain inference computes the domain of relevant relations. For instance, for an array a, the domain of an index variable (i.e., an x used in an expression a[x]) is the set 0 to a.length – 1 (inclusive). We use domain inference only on expressions where such a binding is intuitive and obvious—i.e., for predicates set.contains(x), array[x], and map.get(x). Domain inference extracts all syntactic occurrences of such subterms for quantified variables, including any dependencies: for example, when processing

```
forall x: forall int[] a: b.contains(a) && a[x] > 0
```

we must make sure to extract all viable a in order to determine bindings for b. When there are multiple matching subterms, such as

```
forall x: a[x] > 0 && b[x] > 0
```

the domain is the union of all possible domains, in this case the union of the index domains of a and b.

It is a static error whenever there is no such domain. In practice, this occurs precisely whenever the user omits an explicit type for a quantified variable x and in the body of the quantification this variable never appears as an index, key, or set element, or any such element, or only occurs in a context that depends on x itself, such as x.contains(x).

Exceptions. Exceptions in a query body are propagated to the outside. The language guarantees no order in which exceptions are delivered.
Queries such as

```
query(Map.get(int a) == int b) : a == 1 && range(1,2).contains(b)
```

(which compute multiple mappings for a map key) are invalid and raise a query failure through the edu.umass.pql.AmbiguousMapException.

PQL performs boxing/unboxing conversions implicitly, and failed conversions (i.e., attempts to unbox **null**) raise an exception.

Equality. We allow both the == operator and the = operator for equality comparison (albeit at different operator precedence, following the Java language definition).

Equality is reference equality, except in the following two situations:

- Equality between strings is always equality via equals (value equality).
- Objects in a map or set are considered equal under the terms of the dynamic map or set type. For example, keys of a java.util.HashMap are equal iff they are equal in terms of equals.

User-Defined Reductions. We allow user-defined reductions. A reduction operator r must have the following properties:

- r must be a static method with signature **public static** T r(T, T). The type T must be compatible with the values being reduced and determines the result type of the reduction.
- The type T must be unambiguous. This requirement is relevant when r is overloaded and type inference cannot determine a unique T.

- r must be associative. We distribute the reduction phase by having each core run reductions on part of the data, then merge the results of individual cores. Hence, our runtime system partitions reductions based on what hardware the program is run on. If r is not associative, we may get wrong results.
- r must be commutative. This is not technically required by our current implementation, as it merges the data in order. However, future implementations may choose other reduction orders.
- r must have a neutral element. If T is a reference type, the neutral element is always **null**. For primitive types, r must have a special-purpose Java annotation that specifies the neutral element.

User-defined reduction operators can violate both the parallelization properties (since their execution is sequential and can take arbitrarily long, although multiple are run in parallel) and the declarativeness of PQL. The language blindly trusts that such reductions respect the above stated properties, and chooses an evaluation plan based on this assumption.

3 Implementation and Optimization

Our current prototype implementation of PQL consists of a front-end compiler (integrated with the Oracle `javac`). Our implementation generates ASTs directly and leaves it to the `javac` backend to generate bytecode (which the JIT compiler may optimize further). The runtime support is currently entirely library-based, requiring no VM changes. However, specialized VM support can enable higher levels of optimization in the future. (E.g., by storing fields in random-access tables instead of contiguous objects, it may be more profitable to perform a query using such tables than using other participating relations.) In order to anticipate different back-ends in the future, and also to isolate the front-end language from back-end optimization techniques, we have introduced an intermediate language, PQIL. PQIL is a complex IL—here we discuss its essence and design rationale.

PQL is translated into PQIL and optimized using relational optimization techniques (similarly to an SQL query optimizer), with emphasis on interfacing with Java data and on parallelization. The minimal independent code unit in PQIL is a "join", which can be either a primitive join or a control structure. Control structures allow the sequencing of joins (i.e., conjunction) or combination of all their alternatives (disjunction). Consider the example query of Section 2.2 for identifying a three-character word in a set of strings. We repeat the query below for ease of reference:

```
result = query(Set.contains(byte[] ba)):
           exists i: data_array[i] == ba
             && exists j: range(0, RECORD_SIZE − 3).contains(j)
               && ba[j] == ((byte)'0') && ba[j+1] == ((byte)'1') && ba[j+2] == ((byte)'2');
```

This query is translated to the following PQIL representation:

```
Reduce[SET(?ba): ?result]: {
  JAVA_TYPE<byte[]>(?ba);
  ARRAY_LOOKUP_Object(?data_array; ?i, ?ba);
  ARRAY_LOOKUP_Byte(?ba; ?j, '0');
```

```
    ADD_Int(?j, 1; ?i8);
    ARRAY_LOOKUP_Byte(?ba; ?i8, '1');
    ADD_Int(?j, 2; ?i10);
    ARRAY_LOOKUP_Byte(?ba; ?i10, '2');
    INT_RANGE_CONTAINS(0, (RECORD_SIZE – 3); ?j);
}
```

The above is a reduction over the result of a series of joins that we consider in conjunction. Question marks as prefix indicate variables (named and temporary), all other identifiers are constants of one form or another. As can be seen, PQIL contains predefined predicates for all primitive expressions of the PQL language, turning evaluation of Q-expressions into relational joins. There are some 100 predefined predicates in total, covering types (e.g., JAVA_TYPE<**byte**[]> above), arithmetic (e.g., Add_Int, BITOR_Int), comparisons (e.g., LTE_Int), array/set/map lookup operations (CONTAINS, INT_RANGE_CONTAINS, ARRAY_LOOKUP_OBJECT), and more.

At the outermost level, the translation of a PQL query always contains a reduction operation. Operations forall and exists, as well as constructing maps, sets, etc. are all translated into special-purpose reduction operators. In the above PQIL code, the first line indicates the reduction: we compute a set of all ba, which we write into a result. To compute ba, we consider the body: we locate all ba that satisfy the conjunction of all conditions specified below:

- "JAVA_TYPE<**byte**[]>(?ba);": correct type.
- "ARRAY_LOOKUP_Object(?data_array; ?i, ?ba);": contained in data_array at index i (a variable that has no further purpose).
- "ARRAY_LOOKUP_Byte(?ba; ?j, '0');": at index j, ba has the character '0'.
- "ADD_Int(?j, 1; ?i8);": j+1 = i8.
- "ARRAY_LOOKUP_Byte(?ba; ?i8, '1');": at index i8, ba has character '1'.
- "ADD_Int(?j, 2; ?i10);": j+2 = 10.
- "ARRAY_LOOKUP_Byte(?ba; ?i10, '2');": at index i10, ba has character '2'.
- "INT_RANGE_CONTAINS(0, (RECORD_SIZE – 3); ?j);": j is contained in the range 0 through RECORD_SIZE – 3.

Here, RECORD_SIZE – 3 is a Java expression— RECORD_SIZE is not a query variable. Just as any other Java expression (however complex), it is a constant from the perspective of the query. Our query execution mechanism ensures that we compute the Java expression's value only once and cache it for the rest of query execution. (Recall that we make the assumption that queries do not have side effects, which is true modulo a few remaining effects "by design" such as OutOfMemoryExceptions, and user-defined reductors, which may violate our assumptions.)

The above list captures all constraints we want to have on ba, but the order of joining the 8 relations is crucial for performance. The order is determined by "access path selection", a standard optimization from the database literature [14], to estimate the cheapest way to join the predicates together. The overall process involves a number of steps. We begin with domain inference, which in this case has no effect: no more restrictions on the values of the existential variable can be inferred. Next, we determine dependencies between joins and eliminate unused variables (such as i in our example, transforming ARRAY_LOOKUP_Object(?data_array; ?i, ?ba); to

ARRAY_LOOKUP_Object(?data_array; _, ?ba);). Finally, we compute an access plan, i.e., the order in which the components of the reduction's join should be executed. There are several concerns in selecting this sequence. For instance:

- We reason about indexing: in which order do we bind our variables? Section 2.2 gives high-level examples of this challenge. In our IL, we write J(!x) to indicate that J binds variable x. After access path selection, there must be no free variables remaining.
- We avoid relations that cannot be traversed in the current implementation. For instance, in the above PQIL block, the predicate JAVA_TYPE<**byte**[]>(?ba); cannot form the beginning of the traversal: we have no way to enumerate all **byte**[] objects on the program heap without VM support, even if this were the most efficient way to evaluate the query. Therefore, the predicate can only appear at a position in the sequence where variable ba has been bound.
- Even if a relation can be traversed, it may not be parallelizable. For parallelization, each in-memory join object can expose an interface that allows (depending on the join itself) reductions such as our set computation to perform random access into the join. Queries that can be executed in parallel get a significant bonus during access path selection.
- We translate the query access plan into nested Java loops for later execution inside the VM (after JIT compilation and VM optimization of the bytecode). In practice, the outer loop is often parallel, hence it is important to have it be over a relation large enough to be profitably partitioned. It is also important to ensure good locality (e.g., accesses to consecutive, related data) for the computations in inner loops.

We represent each join in our intermediate language as a join object. There are three classes of join objects: primitive joins (such as ARRAY_LOOKUP_Object), composite joins (e.g., the above block of atomic joins, or the reduction), and "custom joins", which represent the custom generated code in the form of nested Java loops that we just described. This design gives us flexibility since it isolates the optimization logic from the runtime system implementation.

Note that our current implementation makes all scheduling decisions statically. In general, this may not be optimal; there are cases where dynamic information can make a significant difference between picking one option or another. Such dynamic information concerns mainly the size of relations and the distribution of values in a map or set (i.e., the likelihood of that a key will return a value, which determines the selectivity of a join). For example, when joining two maps over their key, it is preferable to iterate over the smaller map in the outer loop and perform lookups on the bigger map. PQIL has explicit primitives %SELECT_PATH and %SCHEDULE to designate that the sequencing of joins in a block is to occur at a later phase and what information it can use. We currently do not use the dynamic access path selection facility because it only works with interpreted execution of our intermediate language (i.e., not with the nested loops execution model).

The PQIL implementation also integrates several optimizations. These include simple optimizations, such as elimination of redundant joins (e.g., two occurrences of the same primitive join in the same conjunctive block, or a type check that is statically known to be true, such as the JAVA_TYPE<**byte**[]>(?ba) in our earlier example), or unification of joins (e.g., simplifying LOOKUP(m, k, _) and LOOKUP(m, k, v) to just LOOKUP(m, k, v)). There are also advanced optimizations for flattening and merging nested queries. For a good example, consider the query:

```
query(Map.contains(key) == newset):
  newset == query(Set.contains(value)): array[value] == key;
```

This query inverts the mapping of an array, producing a map where all the array values become keys for the sets of all array indices where each value appears. In PQIL, this translates (after domain inference) into:

```
Reduce[MAP(?key, ?newset):!result]: {
    ARRAY_LOOKUP_Object(?array; _, !key);
    Reduce[SET(?value):!newset]: {
        ARRAY_LOOKUP_Object(?array; !value, ?key);
    }
}
```

which is correct but inefficient: we iterate over all entries in array to bind key, then in the inner reduction iterate over array *again* to find all the values that map to key.

PQIL flattens the nested queries by allowing *nested reductors* and an accompanying optimization. Nested reductors are only usable in the 'value' field of maps and default maps, where they provide a more compact notation for reductions such as the above:

```
Reduce[MAP(?key, SET(?value)):!result]: { ARRAY_LOOKUP_Object(?array; !value, !key); }
```

Note that the variable ?newset has completely vanished from the query. Furthermore, the query (rewritten in this form) can be executed in a single traversal over array, reducing the iteration time from quadratic to linear.

The condition for this optimization to apply is that we have a reduction of the form

```
Reduce[MAP(?key, ?value):!result1] { ...before... Reduce[R:!value]: body ...after... }
```

such that *before* and *after* do not reference value. If this is the case, we rewrite to

```
Reduce[MAP(?key, R):!result1] { ...before... body ...after... }
```

4 Evaluation

We evaluate the efficiency of our prototype PQL implementation by applying it to a number of tasks from prior literature:

- *bonus*, the task of computing the salary bonuses of employees. This is a well-known example from the databases literature, employed, e.g., in Yang et al.'s map-reduce-merge work [17].

  In this task we are given a set of employees such that each employee has an associated department and a set of accumulated bonuses. We compute a map from each employee to the total bonus, modified by a departmental modifier factor.
- *threegrep*, the example discussed in sections 2.2 and 3, of finding all strings (in a set of 100-character strings) that contain the substring "012" [6].
- *webgraph*, a task defined by Yang et al. [17], in which we are given a set of documents and links between them: each document has a set of out-link objects, which identify the origin document and the document it points to. The task here is to identify the set of all documents that point to themselves via one point of indirection.
- *wordcount* is the task of computing the absolute numbers of occurrences of words in documents. We assume that the words have already been tokenized, stemmed etc., and are matched to a unique integer word ID. The result of this computation is a map from word IDs to the number of times they occur in a collection of documents.

We implemented each of the above tasks in several different ways:

- *pql*: As a PQL/Java query.
- *manual*: As a single-threaded Java method.
- *manual-para*: As multi-threaded, hand-optimized Java code.
- *sql*: As an SQL database query together with SQL database table configurations, both for Postgres and MySQL.
- *hadoop*: As a map-reduction running on the Hadoop framework.

SQL and Hadoop are not the most natural points of comparison technology-wise, but in conceptual terms they are the most closely related systems on Java that we are aware of.

In all of our implementations, we made sure to use the same container classes for results and intermediate computations, so as to not bias the evaluation results in that respect. We also used the same sources of data:

- *bonus* uses a set of employees and an array of departments. Bonuses are stored as a set as part of the employee object. *bonus* also stores an array of employees, indexed by employee number, which we use for the SQL and Hadoop implementations as well as for the manually parallelized version.
- *threegrep* uses an array of byte arrays to store the strings we are looking for.
- *webgraph* uses a set of document objects, each of which stores a set of link objects that contain references to the link target object. Document objects have unique IDs and are stored in an array, which we use for communicating with SQL and Hadoop, and for our manually parallelized implementation.
- *wordcount* uses the same representation as *webgraph* for documents, with each document containing an array of integers representing the words in the document body, in sequence.

Our manual and manually parallelized implementations were mostly straightforward, except as we note below. The manual implementations and the PQL implementations were the only ones that consistently used the 'canonical' data representation (i.e., sets for *bonus*, *webgraph*, and *wordcount*). All other implementations had to rely on auxiliary arrays/tables either for communication or for optimization (manually parallelized code). As a result, the SQL implementations shown are often simpler, but only because a single relation combines information from multiple Java data structures. The SQL version becomes significantly more complex if the translation code from Java is included. In contrast, the PQL implementation works directly on the Java structures, provides static type checking, and, arguably, is a better syntactic fit for everyday parallel tasks.

4.1 Benchmark Implementation Details

To understand the details of our benchmark implementations, it is helpful to recall some particulars of the systems we compare to. Hadoop processes all data as $\langle key, value \rangle$ pairs. It first goes through a map phase, which maps $\langle$k,v$\rangle$ to different $\langle$k2,v2$\rangle$, then aggregates all v2 for the same k2 in the reduce phase, which produces $\langle$kOut,vOut$\rangle$ pairs.

SQL databases do not expose direct access to their database internals, but instead use the JDBC interface for database-to-Java connectivity. We made an effort to consistently employ best-practice idioms, such as always using batch updates for setting up and prepared statements for updates and queries.

Bonus. The PQL query for *bonus* is:

```
result = query(Map.get(employee) == double bonus):
           employees.contains(employee) &&
           bonus == employee.dept.bonus_factor *
                 (reduce(sumDouble) v:
                   exists Bonus b: employee.bonusSet.contains(b) && v == b.bonus_base);
```

In the SQL implementation, we use three tables (employees, bonuses, departments) with suitable SQL types. The main SQL query is:

```
SELECT employees.eid, SUM(bonuses.base * factor)
  FROM employees
       JOIN departments ON departments.did = employees.dept
       JOIN bonuses ON bonuses.eid = employees.eid
  GROUP BY employees.eid
```

(The SQL code shown here and later intends to illuminate the main component of the benchmark in terms of performance. It is not representative for conciseness comparisons, since it omits large amounts of scaffolding code to convert the data from and to Java structures.)

For the Hadoop implementation, we make the department table available and transmit employees via employee ID, followed by department ID, followed by a sequence of bonus values. (All these form the map key, with a constant value—we encode the set of the other implementations into a Hadoop map.) The mapper sums up bonus values, looks up the department and multiplies. The reducer merely aggregates.

In our manually parallelized version of this benchmark we pre-allocated a result hash map to a sufficient size to avoid having to resize the map. Furthermore, we arranged for the employee hash method to map each key to a unique bucket in the table. This allowed us to have our parallel threads write to the same result table without contention.

Threegrep. We have already seen the PQL query for *threegrep* in Section 2.2. For the SQL implementation, data are stored in single table, as string ID (for communicating with Java heap) and char string. The core query is (for Postgres, with a slight difference for MySQL):

```
SELECT id FROM data WHERE (POSITION ("012" IN BODY) > 0)
```

This is a particularly friendly benchmark for SQL implementations. Once the data setup is complete, the above query is quite simple, with explicit support in the language for substring matching.

The Hadoop implementation uses an input map with key-value pairs of the form ⟨string-id:int, string:byte string⟩. The mapper outputs IDs of matching strings, and the aggregator does a straightforward aggregation.

In our manually parallelized implementation we used a synchronized result table shared among the worker threads.

Webgraph. The PQL query for *webgraph* is:

```
result = query(Set.contains(Webdoc doc)):
           documents.contains(doc) &&
           exists link: doc.outlinks.contains(link) &&
             exists link2: link.destination.outlinks.contains(link2) && link2.destination == doc;
```

benchmark	size (# objects)	Lines of Java code				
		manual	manual-parallel	Hadoop	SQL	PQL
bonus	2,360,000	9	50	130	48	8
threegrep	800,000	9	46	60	21	6
webgraph	92,000,000	13	50	105	39	4
wordcount	92,000,000	8	98	93	38	4

Fig. 2. Summary of our experimental setup, including heap size (approximate, due to randomization) and non-comment non-whitespace lines of Java code, excluding syntactic overhead from our benchmarking and import declarations, but including any encoding or decoding overhead required by the framework

The SQL implementation consists of three tables: webdocs (with IDs), links (source ID, target ID, link ID), and words (owner-webdoc-id, offset-in-doc, word-id). (The words table is used for the next benchmark, *wordcount*.) The SQL query is:

```
SELECT links.source FROM links
  JOIN links as links2
    ON links.destination = links2.source AND links2.destination = links.source;
```

For Hadoop, we provide tuples ⟨document-id, array-of-referenced-documents⟩ to the mapper, i.e., we flatten the link set to become part of the document during preprocessing. As output, the mapper produces one pair of each for ⟨src-doc, target-doc⟩ as well as ⟨target-doc, −1− src-doc⟩. (The use of negative 'id's serves to encode inlinks and outlinks differently.) The reducer aggregates all data for the same document and stores everything in a hashmap. If the reducer encounters both a link to 'd' and a link to '−1− d', it knows that we have a circle through 'd' and emits the current document.

Wordcount. The PQL query for *wordcount* is:

```
result = query(Map.get(int word_id) == int idf default 0):
           idf == reduce(sumInt) one over doc:
             one == 1 && documents.contains(doc) && exists i: doc.words[i] == word_id;
```

The SQL setup is the same as in *webgraph* but only uses the words table in a simple query:

```
SELECT word, COUNT(docid) FROM words GROUP BY word
```

The above illustrates our earlier point about PQL using the canonical representations of data, while the SQL and Hadoop implementations can use auxiliary data structures. The words table combines both the documents and the words structures in the PQL implementation. This representational simplification permits a very concise SQL query, but comes at a cost in run-time and code size during setup.

The Hadoop implementation is straightforward, with the mapper input in the form ⟨doc-id, words-as-int-array⟩, mapper output as ⟨word-id, counts-of-word-in-doc⟩, and reducer output ⟨word-id, aggregate-counts-of-word⟩. Figure 2 summarizes our experimental setup. We note that PQL is significantly more compact than any of the parallel alternatives, even though we often broke up our query expressions generously across multiple lines of code, for readability. We did not do the same for SQL: since SQL statements are encoded as strings in Java, formatting them is inconvenient.

4.2 Configuration

We ran our experiments on a Sun SPARC64 (Sparc v9) Enterprise-T5120 system, with 8 cores at 8 SMT threads each, and a dual 6-core Intel Xeon X5650 machine. As runtime we used the native Java 1.7.0-01 on Sun and 1.6.0_26-b03 on Intel, configured to pre-allocate 2 GiB of RAM. For our Hadoop experiments we used the most recent release, Hadoop 0.20.205.0. Our SQL experiments we conducted both on PostgreSQL 8.3.1 and MySQL 5.1.46. The databases were set up to run locally, with no special tuning parameters except as mentioned in the previous section.

PQL Compiler and Runtime. We configured our PQL runtime for its default execution model. In this setup, our runtime executes the body of a reduction in parallel, with each evaluation thread processing a slice of the index space the reduction body has to iterate over. Once a thread has finished its own slice of the 'embarrassingly parallel' initial computation, it follows a pre-assigned merge procedure in which it waits for and merges the results of other threads, in a binary tree fashion: on every round, each thread synchronizes with one other thread to merge their results, and one of them goes on to the next round for further merging. This guarantees that there are precisely n merge steps. We use no further synchronization.

We also enabled all optimizations in our compiler, specifically redundant join elimination, access path selection, and the nested join optimization and translation to Java code (rather than interpretation).

4.3 Measurement Results

Figures 3 and 4 (for the Intel and Sun architectures, respectively) summarize the running times of our PQL implementation on the four benchmarks. The run-times vary from around 100 ms (threegrep on Intel) to roughly half a minute (webgraph on SPARC). Our graphs only show curves for the PQL, manual (single-threaded) and manual-para (hand-optimized multithreaded) versions. The Hadoop and SQL curves are excluded since they skew the results and interfere with their visualization. Hadoop and SQL consistently suffer from low overall performance due to the cost of transferring data back and forth between different heaps and converting it between representations. For reference, we show, in log-scale, the performance of all six implementations for a single-threaded run (i.e., running on a single core, even for the parallel versions) of *threegrep*, in Figure 5. For this benchmark we reduced the size of *threegrep*'s data to $\frac{1}{10}$th.

It is clear that the SQL and Hadoop implementations do not perform at nearly the same level as techniques for running in the same memory space. For native heap execution, setup and result transfer time are negligible. For Hadoop, setup time is small, since we translate values to efficient binary representations, but for both SQL implementations it is prohibitive, most likely due to the necessity of (de)serializing to and from text. Without this overhead MySQL comes within an order of magnitude (but still several times slower) of our native implementations. We chose *threegrep* as a representative benchmark since it is comparatively small and reports only a few dozen results, permitting negligible result transfer cost (which we do not report separately). Our measurements for larger benchmarks are comparable or worse for SQL, though Hadoop

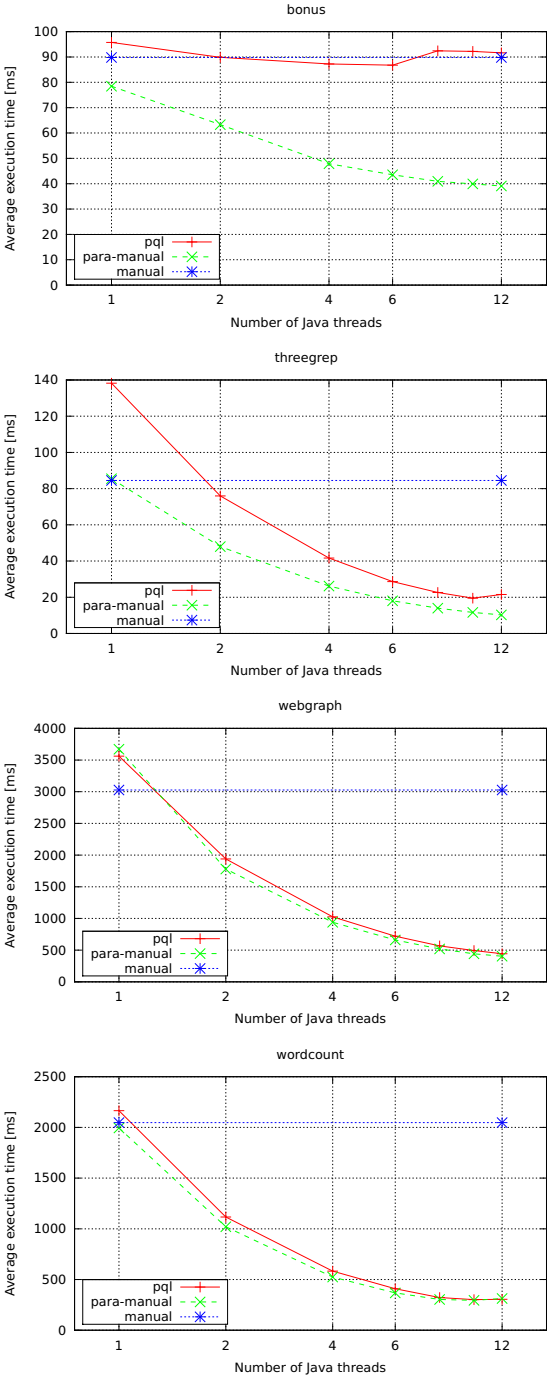


Fig. 3. Results for Intel architecture

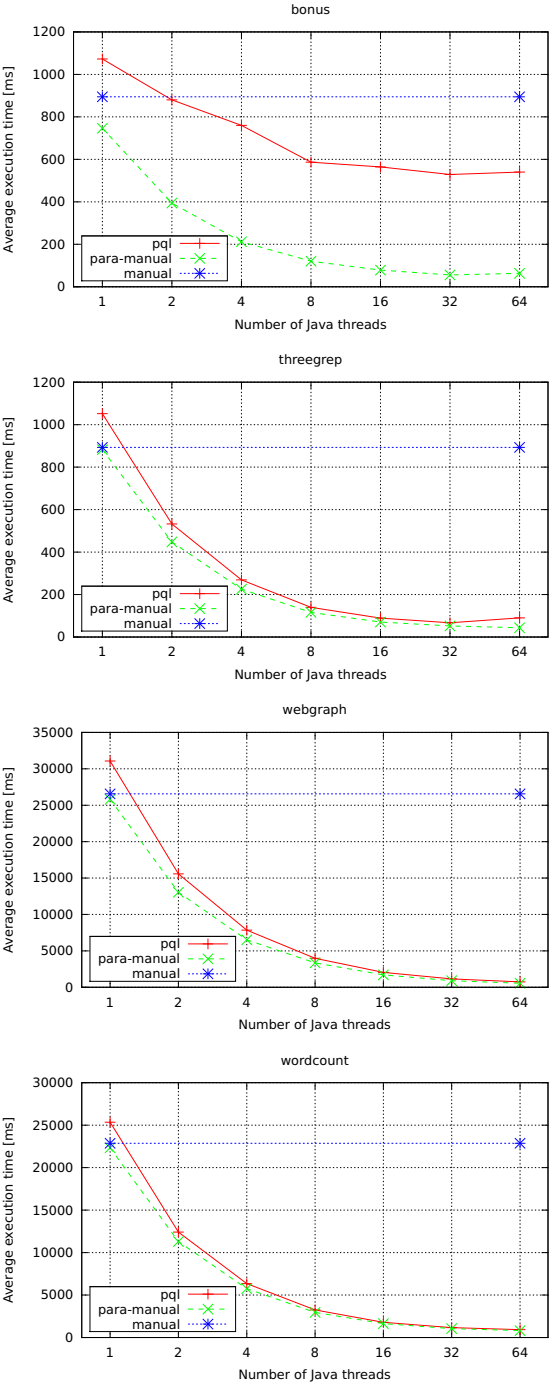


Fig. 4. Results for Sun architecture

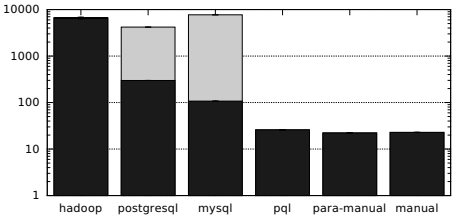


Fig. 5. Results (runtime in ms), in log-scale for all implementations, single-threaded (i.e., on one core only) run of *threegrep* on the Sun architecture. Lighter segments in the bars show setup overheads (i.e., initialization time)—these are large enough to be discernible only for SQL implementations. The figure contains error bars that are non-discernible at this scale

improves (relatively speaking), e.g., Hadoop on regular *threegrep* is "only" 25× slower than our baseline, compared to 84× at reduced benchmark size. Measurements with multithreaded Hadoop show slight improvement, but not enough to be discernible at the scale of Figure 5.

On the more interesting topic of PQL performance compared to manually tuned Java code, we see that the PQL implementation scales in roughly the same patterns as manual code, and nearly matches manual code performance. For *webgraph*, *wordcount*, and *threegrep*, the performance of PQL is strong on both architectures. The remaining benchmark, *bonus*, scales less ideally. The culprit is contention and the high cost of merging results, which dominates the cost of producing the per-processor results. As we described earlier, the manual implementation of *bonus* exploits knowledge about the amount of results it will produce, to pre-allocate a table of the right size and avoid all locking. Such powerful manual optimizations are hard for compilers to reason about or devise. This low-level optimization is particularly important for *bonus* because of its otherwise simple computation. In the PQL implementation, the merge component becomes comparatively bigger than the embarrassingly parallel computation component. At 64 threads, the runtime overhead of a merge is 5.64 for *bonus* (i.e., merge cost is over 5 times as high as computation cost)! (Comparatively, this overhead is 0.01 for *webgraph* and *wordcount*.) The merge overhead thus overrides much of the benefit of parallelization. On Intel, the effect is more pronounced, especially once we exceed 6 threads and start using simultaneous multi-threading.

Wordcount depends critically on our nested join optimization (Section 3): This optimization merges the blocks of joins of inner and outer reduction and thus gives us greater freedom during access path selection. In practice, this allows us to traverse over all documents as the outermost loop, allowing each worker thread to analyze a subset of documents. Without the inner reduction optimization, the only viable outermost loop would be over all array indices in all documents, which means that each thread would have to touch (a slice of) each document. We validated that this rejected access path would parallelize poorly.

For completeness, we list PQL and manually parallelized overhead compared to the baseline, together with the speedups (inverse overhead) observed for PQL, in Figures 6 and 7.

benchmark	Overhead over manual		PQL speedup						
	para-manual	**PQL/1**	**1**	**2**	**4**	**6**	**8**	**10**	**12**
bonus	$1.14 + 0.00\sigma$	$0.94 + 0.00\sigma$	0.94	1.00	1.03	1.03	0.97	0.97	0.98
threegrep	$0.99 + 0.00\sigma$	$0.61 + 0.00\sigma$	0.61	1.11	2.03	2.95	3.73	4.33	3.93
webgraph	$0.82 + 0.00\sigma$	$0.85 + 0.00\sigma$	0.85	1.56	2.95	4.20	5.33	6.12	6.83
idf	$1.03 + 0.00\sigma$	$0.95 + 0.00\sigma$	0.95	1.83	3.51	5.00	6.35	6.78	6.72

Fig. 6. Overhead and speedup measurements on Intel

benchmark	Overhead over manual		PQL speedup						
	para-manual	**PQL/1**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
bonus	$0.83 + 0.00\sigma$	$1.20 + 0.00\sigma$	0.83	1.02	1.18	1.52	1.58	1.69	1.66
threegrep	$0.99 + 0.00\sigma$	$1.18 + 0.00\sigma$	0.85	1.68	3.32	6.39	10.01	13.27	9.91
webgraph	$0.97 + 0.00\sigma$	$1.17 + 0.00\sigma$	0.85	1.71	3.39	6.68	12.98	23.16	35.61
idf	$0.98 + 0.01\sigma$	$1.11 + 0.00\sigma$	0.90	1.84	3.60	7.04	12.78	19.63	24.73

Fig. 7. Overhead and speedup measurements on Sun

Overall, the experiments show how *casual in-memory tasks can benefit from PQL, making seamless declarative parallel processing possible* in the middle of a Java application. Achieving the observed level of performance is the result of significant optimization in the PQL back-end—our original unoptimized implementation (also exploiting parallelism) was more than 10 times slower.

4.4 Discussion

In practice, applications that rely on databases usually store data separately from the Java heap. Doing so in our context would have eliminated the setup cost (though not the query transfer cost). However storing data in databases comes at the price of a semantic gap between Java and the data representation: we cannot add methods to database tables, refactor them in a Java IDE or write unit tests for them easily. The semantic gap extends to the language. We found SQL and PQL to be the languages with the most concise ways to express the computations we were interested in. However, SQL operates on database tables, not sets, maps, objects, and arrays; we thus found it to be an imperfect match for the queries we wished to express. As we saw in Table 2, the programmatic cost of bridging this semantic gap can be considerable. PQL/Java avoids the gap altogether, making declarative parallel programming easy for everyday tasks.

We found (not unexpectedly, but to a larger degree than expected) that Hadoop is not designed for data processing at such a (comparatively) fine-grained scale, i.e., for data that fits into a single computer's memory. For such tasks, we found Hadoop's overhead to be prohibitive. The amount of implementation work needed to communicate with Hadoop efficiently was significantly greater than the amount needed for SQL, since not all queries fit obviously into a map-reduce framework (esp. *webgraph*'s).

For a fair comparison, we should note that Hadoop and SQL databases provide additional features, specifically persistence layers, that are beyond the scope of PQL. However, our experiments suggest that programmers who do not need such persistence and

are only interested in efficient, parallelizable queries that fit within the Java heap have much more to gain from PQL than from (mainstream, unspecialized) SQL database engines or Hadoop.

5 Related Work

It is virtually impossible to be comprehensive when describing related work in parallel languages. There have been numerous and quite diverse approaches, spanning multiple decades. Compared to all of them, the distinguishing feature of our work is that it promotes purely declarative extensions for parallelism, yet keeps the close integration between the declarative sub-language and the sequential host language, with both operating on the same data.

In terms of programming model, the PQL/Java approach can be viewed as map-reduce-on-steroids. Map-reduce computations have a simple, fixed structure that is an easy-to-express special case of our declarative language. PQL/Java generalizes this to offer a full logic-based language in which complex program flows can be expressed and automatically parallelized/optimized. For instance, instead of a plain map-reduce-like structure, an application in our system can have a forall-exists-forall structure, examining combinations of existing data structures and not just mapping over a single one. This need has already been identified in the map-reduce domain. For instance, Google recently introduced the FlumeJava library [4], which supports "a pipeline of MapReduces". In terms of control-flow structures, this is again a special case of our declarative language: the parallel program structures expressible in FlumeJava can also be expressed in PQL/Java. Furthermore, our approach has a much higher-level nature, as it allows aggressive automatic optimization—a direction that FlumeJava begins to pursue with fusion-like parallel loop optimizations, but cannot exploit to nearly the same extent. Of course, directly comparing to specific map-reduce facilities is not appropriate, because the focus of our work is quite different: PQL/Java only targets shared-memory parallelism,[5] while map-reduce libraries are distinguished by their support for distributed, fault-tolerant parallel computations.

Relational databases also have the declarative flavor of the PQL/Java approach, and there is intense research and practical interest in integrating support for relational queries into mainstream programming languages. Microsoft's LINQ and its parallel version, PLINQ [8], are some of the best known such facilities. We already discussed how the design of PLINQ is explicitly not as declarative as that of PQL/Java. Furthermore, we believe that SQL-like syntax is a mismatch for general purpose parallelism: expressing an arbitrary computation with SQL operators such as select, project, join, and difference is awkward. In contrast, we offer a language that has a much more explicit looping structure (forall and exists loops), and an optimizer that leverages the accumulated knowledge from database optimizers, while also understanding the structure of first-order logic sentences. Finally, mainstream relational database engines, such as

[5] The language is designed with the prospect of distributed execution in mind, for future incarnations. The current implementation is for shared-memory machines, however. Other parallel languages—e.g., Fortress [1]—follow the same pattern of starting from shared-memory but designing with an eye for distribution as well.

MySQL and Postgres, do not offer parallelization of a single query, although they support parallel execution of separate queries. This is another example of how applying relational techniques to the usual objects of a Java heap decisively changes the language implementation tradeoffs: Intra-query parallelization makes sense for read-only in-memory data, but less so for traditional transactions in a real database.

In terms of language support for parallelism, there is a multitude of designs that are impossible to cover exhaustively, but follow lines quite distinct from our work. These designs can be as simple as libraries for task-parallelism (e.g., offering a "parallel-for" primitive [12]) and as complex as entire languages for matrix computations, media processing, stream processing, etc. [15,3,10]. Compared to the former, our approach aims to be higher-level, due to the declarativeness of parallel computation. That is, task-parallel libraries only hide the specific mechanisms for parallelism but do not otherwise help address the inherent difficulty of parallel programming. The user is still burdened with structuring the parallel program and little optimization takes place automatically, unlike in PQL/Java. Compared to domain-specific mechanisms for parallelism, our approach is explicitly unifying, with a general declarative language for a substantial subset of all parallel programming tasks.

Finally, PQL/Java is conceptually related to languages that emphasize concurrency and avoid imperative features. It is not a new observation that declarativeness is a good match for parallelism. For instance, the "Declarative Aspects of Multicore Programming" (DAMP) workshop has been held since 2006 and has hosted the presentation of several approaches relating to declarative support for concurrency. Past approaches, however, are typically less general or less declarative than our pure logic-based approach—we offer the first approach that is completely declarative (based on first-order logic, which is truly a specification language, without order dependencies and side-effects), general (can express in a single language the parallel elements of programs from different domains), and unified with a sequential language in a way directly inspired by complexity theory. For a representative comparison, Erlang [2] is a celebrated success story of declarative languages in parallel programming. Nevertheless, Erlang is a Turing-complete and not purely declarative language. (E.g., it is not the case that clauses can be freely reordered without affecting program meaning.) This means that, although Erlang program components communicate asynchronously and, thus, can be easily run in parallel, the responsibility for structuring the program is left with the programmer. Similar comments apply to most high-level languages explicitly designed with parallelism in mind, such as Fortress [1] and X10 [5]. (As with map-reduce mechanisms, however, X10 explicitly targets the much harder problem of distributed execution, while we focus on shared-memory parallelism.) In contrast to such work, our declarative approach consists of specifying in queries what is to be done in parallel but not the exact parallel program structure. The terms of such queries can be freely reordered, factored, and aggressively optimized by the runtime system. Also, the runtime system is fully responsible for deciding what constitutes a task that gets assigned to a processor, unlike in Erlang, where this is dictated by program structure. In short, it is important that our declarative language is not by itself a full, Turing-complete language: the potential for automatic optimization and parallelization is much higher.

We consider this feature crucial for getting higher-level programmability and addressing the challenges of parallel programming.

6 Conclusions

We presented PQL/Java: an approach to parallel programming that employs a purely declarative sublanguage for parallelism, integrated with a mainstream language for sequential computation. PQL queries operate over regular Java data and get automatically optimized by exploiting the declarativeness of the specification. PQL is not a full programming language but it is well-suited for combining, filtering, and reducing large data structures, in a control flow that generalizes map-reduce patterns. Because PQL is a general logic, we expect its users to find innovative ways to express interesting computations, beyond the motivating examples of the original design, fulfilling the promise of a truly high-level, programmer-friendly parallel sublanguage.

Acknowledgments. We would like to thank the anonymous ECOOP reviewers for their feedback. This work was funded by the National Science Foundation under grants CCF-0917774, CCF-0934631, and CCF-1115448.

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, G., Tobin-Hochstadt, S.: The Fortress Language Specification. Technical report, Sun Microsystems (2008)
2. Armstrong, J.: A history of Erlang. In: HOPL III: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages (2007)
3. Catanzaro, B., Fox, A., Keutzer, K., Patterson, D., Su, B.-Y., Snir, M., Olukotun, K., Hanrahan, P., Chafi, H.: Ubiquitous parallel computing from Berkeley, Illinois, and Stanford. IEEE Micro 30(2), 41–55 (2010)
4. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines. In: Programming Language Design and Implementation, PLDI (2010)
5. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2005)
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Operating Systems Design & Implementation (OSDI) (2004)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
8. Duffy, J.: A query language for data parallel programming: invited talk. In: Declarative Aspects of Multicore Programming Workshop, DAMP (2007)
9. Duffy, J., Essey, E.: Parallel LINQ: Running queries on multi-core processors. MSDN Magazine (2007)
10. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS-XII: Architectural Support for Programming Languages and Operating Systems (2006)
11. Immerman, N.: Descriptive Complexity. Springer (1998)

12. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2009)
13. Reichenbach, C., Immerman, N., Smaragdakis, Y., Aftandilian, E.E., Guyer, S.Z.: What can the GC compute efficiently? A language for heap assertions at GC time. In: Object Oriented Programming Systems, Languages, and Applications, OOPSLA (2010)
14. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 23–34 (1979)
15. Snyder, L.: The design and development of ZPL. In: HOPL III: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages (2007)
16. Willis, D., Pearce, D.J., Noble, J.: Efficient Object Querying for Java. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 28–49. Springer, Heidelberg (2006)
17. Yang, H.-C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: ACM SIGMOD Int. Conf. on Management of Data (2007)

Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?

Stas Negara, Mohsen Vakilian, Nicholas Chen,
Ralph E. Johnson, and Danny Dig

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{snegara2,mvakili2,nchen,rjohnson,dig}@illinois.edu

Abstract. Researchers use file-based Version Control System (VCS) as the primary source of code evolution data. VCSs are widely used by developers, thus, researchers get easy access to historical data of many projects. Although it is convenient, research based on VCS data is incomplete and imprecise. Moreover, answering questions that correlate code changes with other activities (e.g., test runs, refactoring) is impossible.

Our tool, CodingTracker, non-intrusively records fine-grained and diverse data *during* code development. CodingTracker collected data from 24 developers: 1,652 hours of development, 23,002 committed files, and 314,085 testcase runs.

This allows us to answer: How much code evolution data is not stored in VCS? How much do developers intersperse refactorings and edits in the same commit? How frequently do developers fix failing tests by changing the test itself? How many changes are committed to VCS without being tested? What is the temporal and spacial locality of changes?

1 Introduction

Any successful software system continuously evolves in response to ever-changing requirements [35]. Developers regularly add new or adjust existing features, fix bugs, tune performance, etc. Software evolution research extracts the code evolution information from the system's historical data. The traditional source of this historical data is a file-based Version Control System (VCS).

File-based VCSs are very popular among developers (e.g., Git [20], SVN [47], CVS [7]). Therefore, software evolution researchers [1, 10, 11, 13, 14, 16–19, 21, 22, 24, 27, 28, 31, 34, 40, 45, 46, 49–51, 54] use VCS to easily access the historical data of many software systems. Although convenient, using VCS code evolution data for software evolution research is inadequate.

First, it is *incomplete*. A single VCS commit may contain hours or even days of code development. During this period, a developer may change the same code fragment multiple times, for example, tuning its performance, or fixing a bug. Therefore, there is a chance that a subsequent code change would override an earlier change, thus *shadowing* it. Since a shadowed change is not present in the

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 79–103, 2012.

code, it is not present in the snapshot committed to a Version Control System (VCS). Therefore, code evolution research performed on the snapshots stored in the VCS (like in [16–18]) does not account for shadowed code changes. Ignoring shadowed changes could significantly limit the accuracy of tools that try to infer the intent of code changes (e.g., infer refactorings [10, 11, 21, 49, 50], infer bug fixes [23, 30, 32, 33, 39, 53]).

Second, VCS data is *imprecise*. A single VCS commit may contain several overlapping changes to the same program entity. For example, a refactored program entity could also be edited in the same commit. This overlap makes it harder to infer the intent of code changes.

Third, answering research questions that correlate code changes with other development activities (e.g., test runs, refactoring) is *impossible*. VCS is limited to code changes, and does not capture many kinds of other developer actions: running the application or the tests, invoking automated refactorings from the IDE, etc. This severely limits the ability to study the code development process. How often do developers commit changes that are untested? How often do they fix assertions in the failing tests rather then fixing the system under test?

Code evolution research studies how the code is changed. So, it is natural to make changes be first-class citizens [42, 44] and leverage the capabilities of an Integrated Development Environment (IDE) to capture code changes online rather than trying to infer them post-mortem from the snapshots stored in VCS. We developed a tool, CodingTracker, an Eclipse plug-in that unintrusively collects the fine-grained data about code evolution of Java programs. In particular, CodingTracker records every code edit performed by a developer. It also records many other developer actions, for example, invocations of automated refactorings, tests and application runs, interactions with VCS, etc. The collected data is so precise that it enables us to reproduce the state of the underlying code at any point in time. To represent the raw code edits collected by CodingTracker uniformly and consistently, we implemented an algorithm that infers changes as Abstract Syntax Tree (AST) node operations. Section 2.1 presents more details about our choice of the unit of code change.

We deployed CodingTracker to collect evolution data for 24 developers working in their natural settings. So far, we have collected data for 1,652 hours of development, which involve 2,000 commits comprising 23,002 committed files, and 9,639 test session runs involving 314,085 testcase runs.

The collected data enables us to answer five research questions:

Q1: How much code evolution data is not stored in VCS?
Q2: How much do developers intersperse refactorings and edits in the same commit?
Q3: How frequently do developers fix failing tests by changing the test itself?
Q4: How many changes are committed to VCS without being tested?
Q5: What is the temporal and spacial locality of changes?

We found that 37% of code changes are *shadowed* by other changes, and are not stored in VCS. Thus, VCS-based code evolution research is incomplete. Second,

programmers intersperse different kinds of changes in the same commit. For example, 46% of refactored program entities are also edited in the same commit. This overlap makes the VCS-based research imprecise. The data collected by CodingTracker enabled us to answer research questions that could not be answered using VCS data alone. The data reveals that 40% of test fixes involve changes to the tests, which motivates the need for automated test fixing tools [8, 9, 36]. In addition, 24% of changes committed to VCS are untested. This shows the usefulness of continuous integration tools [2, 3, 25, 26]. Finally, we found that 85% of changes to a method during an hour interval are clustered within 15 minutes. This shows the importance of novel IDE user interfaces [4] that allow developers to focus on a particular part of the system.

This paper makes the following major contributions:

1. The design of five questions about the reliability of VCS data in studying code evolution. These five research questions have never been answered before.
2. A field study of 24 Java developers working in their natural environment. To the best of our knowledge, this is the first study to present quantitative evidence of the limitations of code evolution research based on VCS data.
3. CodingTracker, an Eclipse plug-in that collects a variety of code evolution data online and a replayer that reconstructs the underlying code base at any given point in time. CodingTracker is open source and available at http://codingspectator.cs.illinois.edu.
4. A novel algorithm that infers AST node operations from low level code edits.

2 Research Methodology

To answer our research questions, we conducted a user study on 24 participants. We recruited 13 Computer Science graduate students and senior undergraduate summer interns who worked on a variety of research projects from six research labs at the University of Illinois at Urbana-Champaign. We also recruited 11 programmers who worked on open source projects in different domains, including marketing, banking, business process management, and database management. Table 1 shows the programming experience of our participants[1]. In the course of our study, we collected code evolution data for 1,652 hours of code development with a mean distribution of 69 hours per programmer and a standard deviation of 62.

To collect code evolution data, we asked each participant to install CodingTracker plug-in in his/her Eclipse IDE. During the study, CodingTracker recorded a variety of evolution data at several levels ranging from individual code edits up to the high-level events like automated refactoring invocations, test runs, and interactions with Version Control System (VCS). CodingTracker employed CodingSpectator's infrastructure [48] to regularly upload the collected

[1] Note that only 22 out of 24 participants filled the survey and specified their programming experience.

Table 1. Programming experience of the participants

Number of participants	Programming Experience (years)
1	1 - 2
4	2 - 5
11	5 - 10
6	> 10

data to our centralized repository. Section 4 presents more details about the data CodingTracker collects.

2.1 Unit of Code Change

Our code evolution questions require measuring the number of code changes. Therefore, we need to define a unit of code change. Individual code edits collected by CodingTracker represent code changing actions of a developer in the most precise way, but they are too irregular to serve as a unit of change in our code evolution analysis. A single code edit could represent typing a single character or inserting a whole class declaration. Moreover, even if several code edits are equivalent in the number of affected characters, they could have a totally different impact on the underlying code depending on whether they represent editing a comment, typing a long name of a single variable, or adding several short statements.

We define a unit of code change as an atomic operation on an Abstract Syntax Tree (AST) node: *add*, *delete*, or *update*, where *add* adds a node to AST, *delete* removes a node from AST, and *update* changes a property of an existing AST node (e.g., name of a variable). We represent a *move* operation, which moves a child node from one parent to another one in an AST, as two consequent AST node operations: *delete* and *add*.

To infer AST node operations from the collected raw edits, we apply our novel inferencing algorithm described in Section 5. Our research questions require establishing how AST node operations correlate with different developer's actions, e.g., whether an AST operation is a result of a refactoring, whether AST operations are followed by a commit or preceded by tests, etc. Therefore, CodingTracker inserts the inferred AST node operations in the original event sequence right after the subsequence of code edits that produce them. We answer every research question by processing the output of the inferencing algorithm with the question-specific analyzer.

3 Research Questions

3.1 How Much Code Evolution Data Is Not Stored in VCS?

A single VCS commit may contain hours or days worth of development. During this period, a developer may change the same code fragment multiple times, with the latter changes *shadowing* the former changes.

Figure 1 presents a code evolution scenario. The developer checks out the latest revision of the code shown in Figure 1(a) and then applies two refactorings and performs several code changes to a single method, `adjust`. First, the developer renames the method and its local variable, `base`, giving them intention-revealing names. Next, to improve the precision of the calculation, the developer changes the type of the computed value and the production factor, `unit`, from `int` to `float` and assigns a more precise value to `unit`. Last, the developer decides that the value of `unit` should be even more precise, and changes it again. Finally, the developer checks the code shown in Figure 1(d) into the VCS.

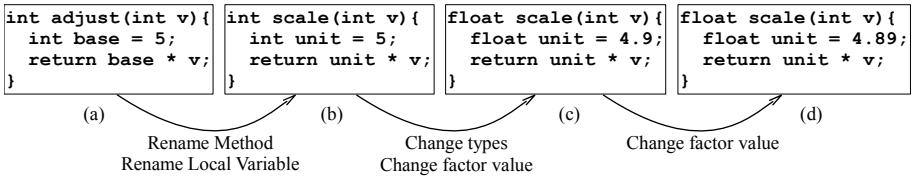


Fig. 1. A code evolution scenario that illustrates a shadowed code change and an overlap of refactorings with other code changes

Note that in the above scenario, the developer changes the value assigned to `unit` twice, and the second change *shadows* the first one. The committed snapshot shown in Figure 1(d) does not reflect the fact that the value assigned to `unit` was gradually refined in several steps, and thus, some code evolution information is lost.

To quantify the extent of code evolution data losses in VCS snapshots, we calculate how many code changes never make it to VCS. We compute the total number of changes that happen in between each two commits of a source code file and the number of changes that are *shadowed*, and thus, do not reach VCS. We get the number of *reaching* changes by subtracting the number of shadowed changes from the total number of changes. To recall, a unit of code change is an *add*, *delete*, or *update* operation on an AST node. For any two operations on the same AST node, the second operation always shadows the first one. Additionally, if an AST node is both added and eventually deleted before being committed, then all operations that affect this node are shadowed, since no data about this node reaches the commit.

Figure 2 shows the ratio of reaching and shadowed changes for our participants. Note that we recorded interactions with VCS only for 15 participants who used Eclipse-based VCS clients. A separate bar presents the data for each such participant. The last bar presents the aggregated result. Overall, we recorded 2,000 commits comprising 23,002 committed files.

The results in Figure 2 demonstrate that on average, 37% of changes are shadowed and do not reach VCS. To further understand the nature of shadowed

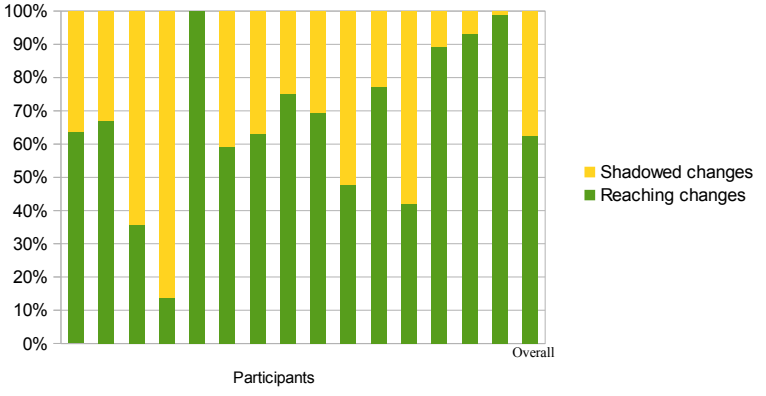


Fig. 2. Ratio of reaching and shadowed changes

code changes, we counted separately those shadowed changes that are commenting/uncommenting parts of the code or undoing some previous changes. If a change is both commenting/uncommenting and undoing, then it is counted as commenting/uncommenting. Figure 3 presents the results. Overall, 78% of shadowed code changes are authentic changes, i.e., they represent actual changes rather than playing with the existing code by commenting/uncommenting it or undoing some previous changes.

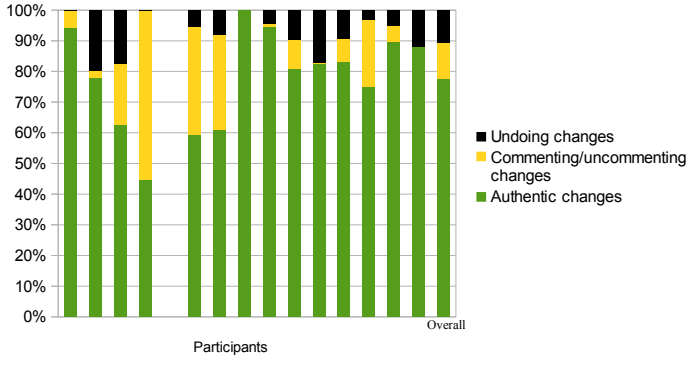


Fig. 3. Composition of shadowed changes. The fifth bar is missing, since there are no shadowed changes for this participant.

Our results reveal that more than a third of all changes do not reach VCS and the vast majority of these lost changes are authentic. Thus, a code evolution analysis based on snapshots from VCS misses a significant fraction of important code changes, which could lead to imprecise results. Further research is required to investigate the extent of this imprecision.

3.2 How Much Do Developers Intersperse Refactorings and Edits in the Same Commit?

Many tools [10, 11, 21, 29, 49, 50] compare a program's snapshots stored in VCS to infer the refactorings applied to it. As the first step, such a tool employs different similarity measures to match the refactored program entities in the two compared snapshots. Next, the tool uses the difference between the two matched program entities as an indicator of the kind of the applied refactoring. For example, two methods with different names but with similar code statements could serve as an evidence of a Rename Method refactoring [11]. If a refactored program entity is additionally changed in the same commit, both matching it across commits and deciding on the kind of refactoring applied to it become harder. Such code evolution scenarios undermine the accuracy of the snapshot-based refactoring inference tools.

Figure 1 shows an example of such a scenario. It starts with two refactorings, Rename Method and Rename Local Variable. After applying these refactorings, the developer continues to change the refactored entities – the body and the return type of the renamed method; the type and the initializer of the renamed local variable. Consequently, versions (a) and (d) in Figure 1 have so little in common that even a human being would have a hard time identifying the refactored program entities across commits.

To quantify how frequently refactorings and edits overlap, we calculate the number of refactored program entities that are also edited in the same commit. Our calculations employ the data collected by CodingTracker for ten participants who both used Eclipse-based VCS clients and performed automated refactorings. Note that we do not consider manual refactorings since they can not be directly captured by our data collector, but rather need to be inferred from the collected data as an additional, non-trivial step.

First, we look at a single kind of program entities – methods. Figure 4 shows the ratio of those methods that are refactored only once before being committed (pure refactored methods) and those methods that are both refactored and edited (e.g., refactored more than once or refactored and edited manually) before being committed to VCS. We consider a method refactored/edited if either its declaration or any program entity in its body are affected by an automated refactoring/manual edit. Figure 4 shows that on average, 58% of methods are both refactored and additionally changed before reaching VCS.

Next, we refine our analysis to handle individual program entities. To detect whether two refactorings or a refactoring and a manual edit overlap, we introduce the notion of a *cluster* of program entities. For each program entity, we compute its cluster as a collection of closely related program entities. A cluster of a program entity includes this entity, all its descendants, its enclosing statement, and all descendants of its enclosing statement, except the enclosing statement's body and the body's descendants. We consider a program entity refactored/edited if any of the entities of its cluster is affected by an automated refactoring/manual edit. Figure 5 demonstrates that on average, 46% of program entities are both refactored and additionally changed in the same commit.

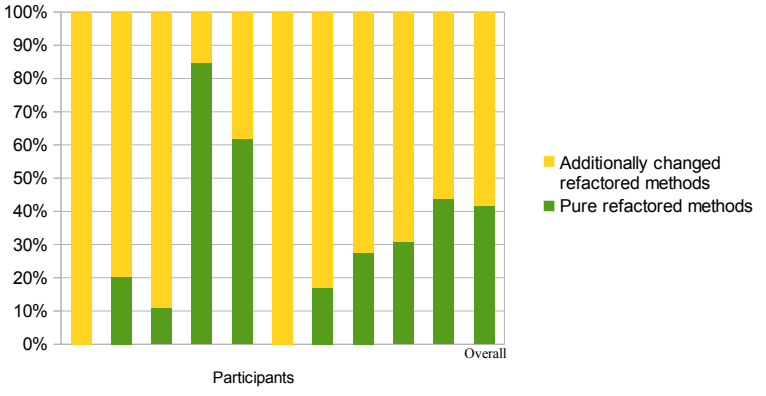


Fig. 4. Ratio of purely refactored methods and those that are both refactored and additionally changed before being committed to VCS

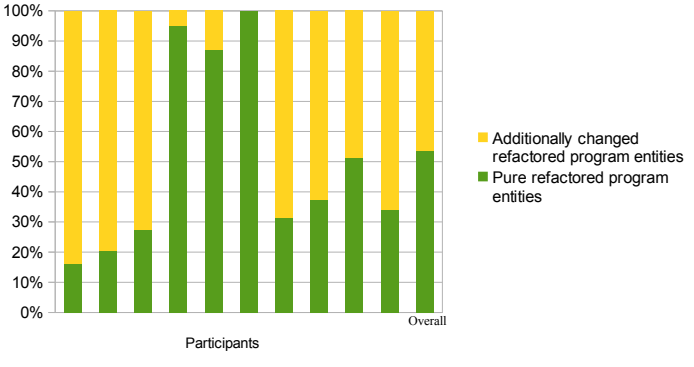


Fig. 5. Ratio of purely refactored program entities and those that are both refactored and additionally changed before reaching a commit

Our results indicate that most of the time, refactorings are tightly intermixed with other refactorings or manual edits before reaching VCS. This could severely undermine the effectiveness of refactoring inference tools that are based on VCS snapshots [10, 11, 21, 29, 49, 50]. Our findings serve as a strong motivation to build a refactoring inference tool based on the precise, fine-grained data collected by CodingTracker and compare its accuracy against the existing snapshot-based tools.

3.3 How Frequently Do Developers Fix Failing Tests by Changing the Test Itself?

In response to ever-changing requirements, developers continuously add new features or adjust existing features of an application, which could cause some unit tests to fail. A test that fails due to the new functionality is considered

broken since making it a passing test requires fixing the test itself rather than the application under test. Developers either have to fix the broken tests manually or use recent tools that can fix them automatically [8, 9, 36].

Figure 6 presents a unit test of a parser. This test checks that the parser produces a specific number of elements for a given input. A new requirement to the system introduces an additional parsing rule. Implementing this new feature, a developer breaks this test, because the number of elements in the same input has changed. Thus, the developer needs to update the broken test accordingly.

```
public void testElementsCount(){
  Parser p = new Parser();
  p.parse(getSource());
  assertEquals(p.getCount(), 5);
}
```

Fig. 6. A unit test of a parser that checks the total number of elements in the parser's result

We justify the need for the automated test fixing tools by showing how often such scenarios happen in practice, i.e., how many failing tests are fixed by changing the test itself. We look for these scenarios in the data collected for 15 participants who ran JUnit tests as part of their code development process. Overall, we recorded 9,639 test session runs, involving 314,085 testcase runs. We track a failing test from the first run it fails until the run it passes successfully. Each such scenario is counted as a test fix. If a developer changes the test's package during this time span, we consider that fixing this failing test involves changing it.

Figure 7 shows the ratio of test fixes involving and not involving changes to the tests. Our results show that on average, 40% of test fixes involve changes to the tests. Another observation is that every participant has some failing tests, whose fix requires changing them. Hence, a tool like ReAssert [9] could have benefited all of the participants, potentially helping to fix more than one third of all failing tests. Nevertheless, only a designated study would show how much of the required changes to the failing tests could be automated by ReAssert.

3.4 How Many Changes Are Committed to VCS without Being Tested?

Committing untested code is considered a bad practice. A developer who commits untested code risks to break the build and consequently, cause the disruption of the development process. To prevent such scenarios and catch broken builds early, the industry adopted continuous integration tools (e.g., Apache Gump [2], Bamboo [3], Hudson [25], and Jenkins [26]), which build and test every commit before integrating it into the trunk. Only those commits that successfully

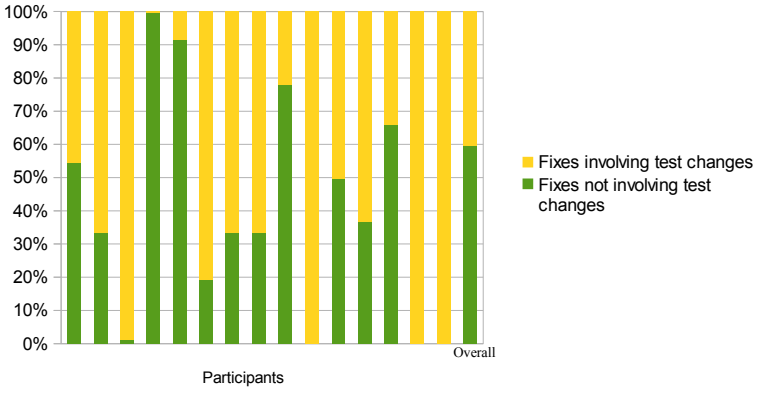


Fig. 7. Ratio of test fixes involving and not involving changes to the tests

pass all the tests are merged into the trunk. Nevertheless, these tools are not pervasive yet. In particular, the majority of the projects that we studied did not employ any continuous integration tools. Therefore, we would like to quantitatively assess the necessity of such tools.

To assess the number of untested, potentially build-breaking changes that are committed to VCS, we measure how much developers change their code in between tests and commits. Our measurements employ the data collected for ten participants who both used Eclipse-based VCS clients and ran JUnit tests. We consider each two consecutive commits of a source code file. If there are no test runs in between these two commits, we disregard this pair of commits[2]. Otherwise, we count the total number of code changes that happen in between these two commits. Also, we count all code changes since the last test run until the subsequent commit as *untested* changes. Subtracting the untested changes from the total number of changes in between the two commits, we get the *tested* changes.

Figure 8 shows the ratio of tested and untested changes that reach VCS. Although the number of untested changes that reach a commit varies widely across the participants, every participant committed at least some untested changes. Overall, 24% of changes committed to VCS are untested. Figure 9 shows that 97% of the untested changes are authentic, i.e., we discard undos and comments.

Note that even a small number of code changes may introduce a bug, and thus, break a build (unless the code is committed to a temporary branch). Besides, even a single developer with a habit to commit untested changes into the trunk may disrupt the development process of the entire team. Thus, our results confirm the usefulness of continuous integration tools, which ensure that all commits merged into the trunk are fully tested.

[2] We are conservative in calculating the amount of untested changes in order to avoid skewing our results with some corner case scenarios, e.g., when a project does not have automated unit tests at all (although this is problematic as well).

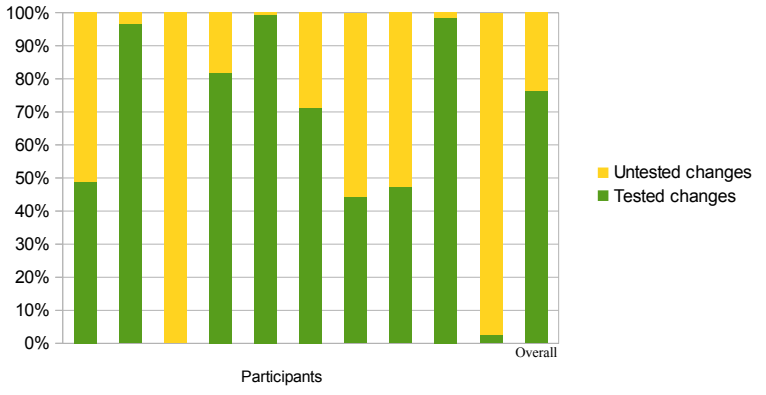


Fig. 8. Ratio of tested and untested code changes that reach VCS

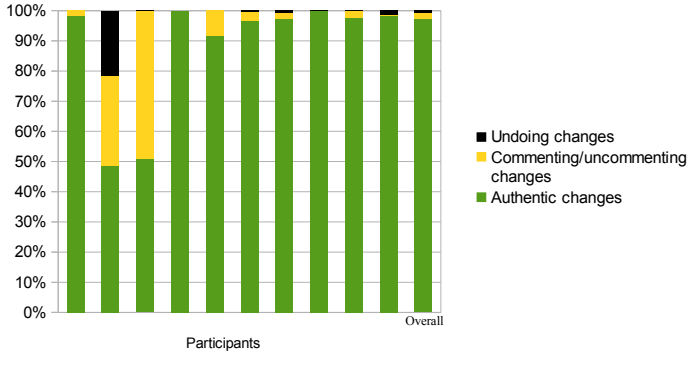


Fig. 9. Composition of untested changes that reach VCS

3.5 What Is the Temporal and Spacial Locality of Changes?

Simplifying the development process and increasing the productivity of a developer are among the major goals of an Integrated Development Environment (IDE). The better an IDE supports code changing behavior of a developer, the easier it is for him/her to develop the code. Code Bubbles [4] is an example of a state-of-the-art IDE with a completely reworked User Interface (UI). The novel UI enables a developer to concentrate on individual parts of an application. For example, a developer could pick one or more related methods that he/she is currently reviewing or editing and focus on them only.

To detect whether developers indeed focus their editing efforts on a particular method at any given point in time, we calculate the distribution of method-level code changes over time. We perform this calculation for all 24 participants who took part in our study, since it does not depend on any particular activity of the participant (e.g., interactions with VCS or test runs). We employ three sliding time windows spanning 15, 30, and 60 minutes. For every code change

that happens in a particular method, we count the number of changes to this method within each sliding window, i.e., the number of changes 7.5 minutes, 15 minutes, and 30 minutes before and after the given change. Then, we sum the results for all code changes of each method. Finally, we add up all these sums for all the methods.

Figure 10 shows the ratio of method-level code changes for each of our three sliding time windows. On average, 85% of changes to a method during an hour interval are clustered within 15 minutes. Our results demonstrate that developers tend to concentrate edits to a particular method in a relatively small interval of time. The implication of this finding is that IDEs should provide visualizations of the code such that a programmer can focus on one method at a time.

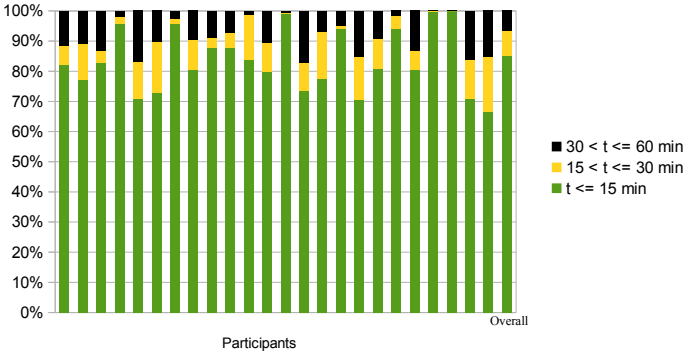


Fig. 10. Ratio of method-level code changes for three sliding time windows: 15, 30, and 60 minutes

4 Collecting Code Evolution Data

To collect code evolution data for our research questions, we developed an Eclipse plug-in, CodingTracker. CodingTracker registers 38 different kinds of code evolution events that are grouped in ten categories. Table 2 presents the complete list of the registered events. CodingTracker records the detailed information about each registered event, including the timestamp at which the event is triggered. For example, for a performed/undone/redone text edit, CodingTracker records the offset of the edit in the edited document, the removed text (if any), and the added text (if any). In fact, the recorded information is so detailed and precise that CodingTracker's replayer uses it to reconstruct the state of the evolving code at any point in time. Note that we need to replay the recorded data to reproduce the actions of a developer since our AST node operations inferencing algorithm is applied offline.

CodingTracker's replayer is an Eclipse View that is displayed alongside other Views of an Eclipse workbench, thus enabling a user to see the results of the replayed events in the same Eclipse instance. The replayer allows to load a recorded sequence of events, browse it, hide events of the kinds that a user is not interested in, and replay the sequence at any desired pace.

Table 2. The complete list of events recorded by CodingTracker

Category	Event	Description
Text editing	Perform/Undo/Redo text edit	Perform/Undo/Redo a text edit in a Java editor
	Perform/Undo/Redo compare editor text edit	Perform/Undo/Redo a text edit in a compare editor
File editing	Edit file	Start editing a file in a Java editor
	Edit unsynchronized file	Start editing a file in a Java editor that is not synchronized with the underlying resource
	New file	A file is about to be edited for the first time
	Refresh file	Refresh a file in a Java editor to synchronize it with the underlying resource
	Save file	Save file in a Java editor
	Close file	Close file in a Java editor
Compare editors	Open compare editor	Open a new compare editor
	Save compare editor	Save a compare editor
	Close compare editor	Close a compare editor
Refactorings	Start refactoring	Perform/Undo/Redo a refactoring
	Finish refactoring	A refactoring is completed
Resource manipulation	Create resource	Create a new resource (e.g., file)
	Copy resource	Copy a resource to a different location
	Move resource	Move a resource to a different location
	Delete resource	Delete a resource
	Externally modify resource	Modify a resource from outside of Eclipse (e.g., using a different text editor)
Interactions with Version Control System (VCS)	CVS/SVN update file	Update a file from VCS
	CVS/SVN commit file	Commit a file to VCS
	CVS/SVN initial commit file	Commit a file to VCS for the first time
JUnit test runs	Launch test session	A test session is about to be started
	Start test session	Start a test session
	Finish test session	A test session is completed
	Start test case	Start a test case
	Finish test case	A test case is completed
Start up events	Launch application	Run/Debug the developed application
	Start Eclipse	Start an instance of Eclipse
Workspace and Project Options	Change workspace options	Change global workspace options
	Change project options	Change options of a refactored project
Project References	Change referencing projects	Change the list of projects that reference a refactored project

To ensure that the recorded data is correct, CodingTracker records redundant information for some events. This additional data is used to check that the reconstructed state of the code matches the original one. For example, for every text edit, CodingTracker records the removed text (if any) rather than just the length of the removed text, which would be sufficient to replay the event. For CVS/SVN commits, CodingTracker records the whole snapshot of the committed file. While replaying text edits and CVS/SVN commits, CodingTracker checks that the edited document indeed contains the removed text and the committed file matches its captured snapshot[3].

Eclipse creates a refactoring descriptor for every performed automated refactoring. Refactoring descriptors are designed to capture sufficient information to enable replaying of the corresponding refactorings. Nevertheless, we found that some descriptors do not store important refactoring configuration options and thus, can not be used to reliably replay the corresponding refactorings. For example, the descriptor of Extract Method refactoring does not capture information about the extracted method's parameters [5]. Therefore, besides recording refactoring descriptors of the performed/undone/redone automated Eclipse refactorings, CodingTracker records the refactorings' effects on the underlying code. A refactoring's effects are events triggered by the execution of this refactoring – usually, one or more events from *Text editing*, *File editing*, and *Resource manipulation* categories presented in Table 2. In a sequence of recorded events, effects of an automated refactoring are located in between its *Start refactoring* and *Finish refactoring* events. To ensure robust replaying, CodingTracker replays the recorded refactorings' effects rather than their descriptors.

5 AST Node Operations Inferencing Algorithm

Our inferencing algorithm converts the raw text edits collected by CodingTracker into operations on the corresponding AST nodes. First, the algorithm assigns a unique ID to every AST node in the old AST. Next, the algorithm considers the effect of each text edit on the position of the node in the new AST in order to match the old and the new AST nodes. The matched nodes in the new AST get their IDs from their counterparts in the old AST. If the content of a matched AST node has changed, the algorithm generates the corresponding *update* operation. The algorithm generates a *delete* operation for every unmatched node in the old AST and an *add* operation for every unmatched node in the new AST, assigning it a unique ID.

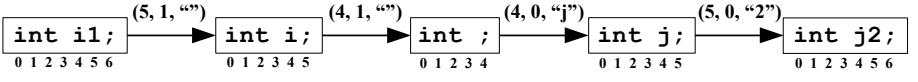
Given an edited document, a single text edit is fully described by a 3-tuple `(<offset>, <removed text length>, <added text>)`, where `<offset>` is the offset of the edit in the edited document, `<removed text length>` is the length of the text that is removed at the specified offset, and `<added text>` is the text that is added at the specified offset. If `<removed text length>` is 0, the edit

[3] Note that replaying CVS/SVN commits does not involve any interactions with Version Control System (VCS), but rather checks the correctness of the replaying process.

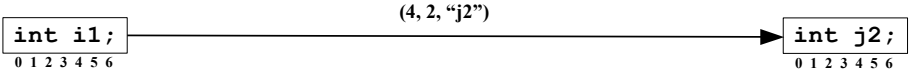
does not remove any text. If `<added text>` is empty, the edit does not add any text. If `<removed text length>` is not 0 and `<added text>` is not empty, the edit replaces the removed text with the `<added text>` in the edited document.

In the following, we describe several heuristics that improve the precision of our inferencing algorithm. Then, we explain our algorithm in more details and demonstrate it using an example.

Gluing. Figure 11(a) illustrates an example of text edits produced by a developer, who renamed variable `i1` to `j2` by first removing the old name using backspace and then typing in the new name. This single operation of changing a variable's name involves four distinct text edits that are recorded by CodingTracker. At the same time, all these text edits are so closely related to each other that they can be "glued" together into a single text edit with the same effect on the underlying text, which is shown in Figure 11(b). We call such "glued" text edits *coherent text edits* and use them instead of the original text edits recorded by CodingTracker in our AST node operations inferencing algorithm. This drastically reduces the number of inferred AST node operations and makes them better represent the intentions of a developer.



(a) A sequence of text edits recorded by CodingTracker for renaming variable `i1` to `j2`.



(b) A *coherent text edit* that "glues" together all text edits shown in Figure 11(a).

Fig. 11. An example of changing a variable's name represented both as individual text edits recorded by CodingTracker (Figure 11(a)) and as a single *coherent text edit* (Figure 11(b)). Each box shows the content of the edited document. The offset of every document's character is shown under each box. The 3-tuples describing text edits are shown above the arrows that connect boxes.

To decide whether a text edit `e2` should be "glued" to a preceding text edit `e1`, we use the following heuristics:

1. `e2` should immediately follow `e1`, i.e., there are no other events in between these two text edits.
2. `e2` should continue the text change of `e1`, i.e., text edit `e2` should start at the offset at which `e1` stopped.

Note that in the above heuristics `e1` can be either a text edit recorded by CodingTracker or a *coherent text edit*, produced from "gluing" several preceding text edits.

Linked Edits. Eclipse offers a code editing feature that allows simultaneous editing of a program entity in all its bindings that are present in the opened document. Each binding of the edited entity becomes an edit box and every text edit in a single edit box is immediately reflected in all other boxes as well. This feature is often used to rename program entities, in particular to name extracted methods. Since text edits in a single edit box are intermixed with the corresponding edits in other edit boxes, to apply our "gluing" heuristics presented above, we treat edits in each box disjointly, constructing a separate *coherent text edit* for every edit box. When a boxed edit is over, the constructed coherent text edits are processed one by one to infer the corresponding AST node operations.

Jumping over Unparsable Code. Our AST node operations inferencing algorithm processes coherent text edits as soon as they are constructed, except the cases when text edits introduce parse errors in the underlying code. Parse errors might confuse the parser that creates ASTs for our algorithm, which could lead to imprecise inferencing results. Therefore, when a text edit breaks the AST, we postpone the inferencing until the AST is well-formed again. Such postponing causes accumulation of several coherent text edits, which are processed by our inferencing algorithm together. Figure 12 shows an example of a code editing scenario that requires inference postponing. A developer inserts brackets around the body of an `if` statement. The first coherent text edit adds an opening bracket, breaking the structure of the AST, while the second coherent text edit adds a closing bracket, bringing the AST back to a well-formed state. The inference is postponed until the second edit fixes the AST. Note that sometimes we have to apply the inferencing algorithm even when the underlying program's AST is still broken, for example, when a developer closes the editor before fixing the AST. This could lead to some imprecision in the inferred AST node operations, but we believe that such scenarios are very rare in practice. In particular, our personal experience of replaying the recorded sequences shows that the code rarely remains in the unparsable state for long time.

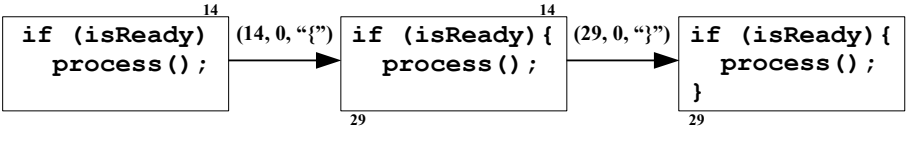


Fig. 12. An example of two code edits: the first edit breaks the AST of the edited program, while the second edit brings the AST back to a well-formed state

Pseudocode. Figure 13 shows an overview of our AST node operations inferencing algorithm. The algorithm takes as input the list of *coherent text edits*, *cteList*, the AST of the edited code before the text edits, *oldAST*, and the AST

after the edits, *newAST*. The output of the algorithm is an unordered collection of the inferred AST node operations.

The inferencing algorithm is applied as soon as a new coherent text edit is completed, unless the underlying code is unparsable at that point, in which case the inferencing is postponed until the code becomes parsable again. As long as the code remains parsable, *cteList* contains a single coherent text edit. If the code becomes unparsable for some time, *cteList* will contain the accumulated coherent text edits that bring the code back into the parsable state. Note that *newAST* represents the code that is a result of the edits in *cteList* applied to the code represented by *oldAST*. Since replaying the edits in *cteList* is not a part of the inferencing algorithm, we supply both *oldAST* and *newAST* as the algorithm's inputs.

Each inferred operation captures the persistent ID of the affected AST node. Persistent IDs uniquely identify AST nodes in an application throughout its evolution. Note that given an AST, a node can be identified by its *position* in this AST. A node's position in an AST is the traversal path to this node from the root of the AST. Since the position of an AST node may change with the changes to its AST, we assign a unique persistent ID to every AST node and keep the mapping from positions to persistent IDs, updating it accordingly whenever a node's position is changed as a result of code changes.

Most of the time, edits in *cteList* affect only a small part of the code's AST. Therefore, the first step of the algorithm (lines 3 – 5) establishes the root of the changed subtree – a *common covering node* that is present in both the old and the new ASTs and completely encloses the edits in *cteList*. To find a common covering node, we first look for a local covering node in *oldAST* and a local covering node in *newAST*. These local covering nodes are the innermost nodes that fully encompass the edits in *cteList*. The common part of the traversal paths to the local covering nodes from the roots of their ASTs represents the position of the common covering node (assigned to *coveringPosition* in line 3).

Next, every descendant node of the common covering node in the old AST is checked against the edits in *cteList* (lines 6 – 18). An edit does not affect a node if the code that this node represents is either completely before the edited code fragment or completely after it. If a node's code is completely before the edited code fragment, the edit does not impact the node's offset. Otherwise, the edit shifts the node's offset with `<added text length> - <removed text length>`. These shifts are calculated by *getEditOffset* and accumulated in *deltaOffset* (line 12). If no edits affect a node, the algorithm looks for its matching node in the new AST (line 15). Every matched pair of nodes is added to *matchedNodes*.

In the following step (lines 19 – 25), the inferencing algorithm matches yet unmatched nodes that have the same AST node types and the same position in the old and the new ASTs. Finally, the algorithm creates an *update* operation for every matched node whose content has changed (lines 26 – 30), a *delete* operation for every unmatched node in the old AST (lines 31 – 33), and an *add* operation for every unmatched node in the new AST (lines 34 – 36).

```
input: oldAST, newAST, cteList // the list of coherent text edits
output: astNodeOperations
1  astNodeOperations = ∅;
2  matchedNodes = ∅;
3  coveringPosition = getCommonCoveringNodePosition(oldAST, newAST, cteList);
4  oldCoveringNode = getNode(oldAST, coveringPosition);
5  newCoveringNode = getNode(newAST, coveringPosition);
6  foreach (oldNode ∈ getDescendants(oldCoveringNode)) { // matches outliers
7    deltaOffset = 0;
8    foreach (textEdit ∈ cteList) {
9      if (affects(textEdit, oldNode, deltaOffset) {
10       continue foreach_line6;
11     } else {
12       deltaOffset += getEditOffset(textEdit, oldNode, deltaOffset);
13     }
14   }
15   if (∃ newNode ∈ getDescendants(newCoveringNode) :
       getOffset(oldNode) + deltaOffset == getOffset(newNode) &&
       haveSameASTNodeTypes(oldNode, newNode)) {
16     matchedNodes ∪= (oldNode, newNode);
17   }
18 }
19 foreach (oldNode ∈ getDescendants(oldCoveringNode) :
         oldNode ∉ getOldNodes(matchedNodes)) { // matches same-position nodes
20   oldPosition = getNodePositionInAST(oldNode, oldAST);
21   newNode = getNode(newAST, oldPosition);
22   if (∃ newNode ∈ getDescendants(newCoveringNode) :
                                    haveSameASTNodeTypes(oldNode, newNode) {
23     matchedNodes ∪= (oldNode, newNode);
24   }
25 }
26 foreach ((oldNode, newNode) ∈ matchedNodes) {
27   if (getText(oldNode) ≠ getText(newNode)) {
28     astNodeOperations ∪= getUpdateOperation(oldNode, newNode);
29   }
30 }
31 foreach (oldNode ∈ getDescendants(oldCoveringNode) :
                                        oldNode ∉ getOldNodes(matchedNodes)) {
32   astNodeOperations ∪= getDeleteOperation(oldNode);
33 }
34 foreach (newNode ∈ getDescendants(newCoveringNode) :
                                        newNode ∉ getNewNodes(matchedNodes)) {
35   astNodeOperations ∪= getAddOperation(newNode);
36 }
```

Fig. 13. Overview of our AST node operations inferencing algorithm

Example. Figure 14 illustrates a coherent text edit that changes a variable declaration. Figure 15 demonstrates the inferred AST node operations for this edit. Connected ovals represent the nodes of the old and the new ASTs. Dashed arrows represent the inferred operations. Labels above the arrows indicate the kind of the corresponding operations.

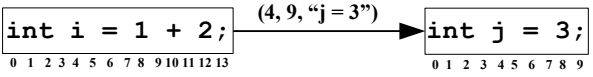


Fig. 14. An example of a text edit that changes a variable declaration

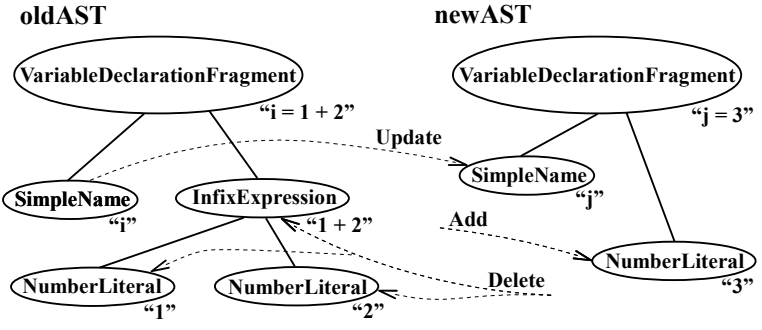


Fig. 15. The inferred AST node operations for the text edit in Figure 14

6 Threats to Validity

There are several factors that might negatively impact the precision of our results. This section discusses the potential influence and possible mitigation for each of these factors.

6.1 Experimental Setup

Issues like privacy, confidentiality, and lack of trust in the reliability of research tools made it difficult to recruit programmers to participate in our study. Therefore, we were unable to study a larger sample of experienced programmers.

Many factors affect programmers' practices. For example, programmers may write code, refactor, test, and commit differently in different phases of software development, e.g., before and after a release. As another example, practices of programmers who work in teams might be different than those who are the sole authors of their programs. Due to the uncontrolled nature of our study, it is not clear how such factors affect our results.

Our participants have used CodingTracker for different periods of time (See Section 2). Therefore, those participants who used CodingTracker more influenced our results more.

Our results are limited to programmers who use Eclipse for Java programming because CodingTracker is an Eclipse plug-in that captures data about the evolution of Java code. However, we expect our results to generalize to similar programming environments.

6.2 AST Node Operations Inferencing Algorithm

To decide whether two individual text edits should be "glued" together, we apply certain heuristics, which are sufficient in most cases. Nevertheless, as any heuristics, they can not cover all possible scenarios. As a result, our algorithm might infer multiple operations for a single change intended by a developer (e.g., a single rename of a variable). This artificial increase in the number of AST node operations can potentially skew the results for each question. However, such corner case scenarios are infrequent and thus, their influence on our results is negligible.

The current implementation of our AST node operations inferencing algorithm does not support the *move* operation, but rather represents the corresponding action as *delete* followed by *add*. Consequently, the number of AST node operations that our data analyzers operate on might be inflated. At the same time, all our results are computed as ratios of the number of operations, which substantially diminishes the effect of this inflation.

Although our AST node operations inferencing algorithm does not expect that the underlying code is always parsable, it produces the most precise results for a particular subsequence of text edits when there is at least one preceding and one succeeding state, in which the code is parsable. The algorithm uses these parsable states to "jump over the gap" of intermediate unparsable states, if any. A scenario without a preceding and succeeding parsable states could cause the algorithm to produce some noise in the form of spurious or missing AST node operations. Such scenarios are very uncommon and hence, their impact on our results is minimal.

7 Related Work

7.1 Empirical Studies on Source Code Evolution

Early work on source code evolution relied on the information stored in VCS as the primary source of data. The lack of fine-grained data constrained researchers to concentrate mostly on extracting high-level metrics of software evolution, e.g., number of lines changed, number of classes, etc.

Eick et al. [13] identified specific indicators for *code decay* by conducting a study on a large (∼100,000,000 LOC) real time software for telephone systems. These indicators were based on a combination of metrics such as number of

lines changed, commit size, number of files affected by a commit, duration of a change, and the number of developers contributing to a file.

Xing et al. [51] analyzed the evolution of design in object-oriented software by reconstructing the differences between snapshots of software releases at the UML level using their tool, UMLDiff. UML level changes capture information at the class level and can be used to study how classes, fields, and methods have changed from each version. From these differences, they tried to identify distinct patterns in the software evolution cycles.

Gall et al. [16] studied the logical dependencies and change patterns in a product family of Telecommunication Switching Systems by analyzing 20 punctuated software releases over two years. They decomposed the system into modules and used their CAESAR technique to analyze how the structure and software metrics of these modules evolved through different releases.

For these kinds of analyses, the data contained in traditional VCS is adequate. However, for more interesting analyses that require program comprehension, relying only on high-level information from VCS is insufficient. In particular, Robbes in his PhD thesis [41, p.70] shows the difference in the precision of code evolution analysis tools applied to fine-grained data vs. coarse-grained VCS snapshots. This client level comparison is complementary to our work, in which we quantify the extent of data loss and imprecision in VCS snapshots independently of a particular client tool.

7.2 Tools for Reconstructing Program Changes

To provide greater insight into source code evolution, researchers have proposed tools to reconstruct high-level source code changes (e.g., operations on AST nodes, refactorings, restructurings, etc.) from the coarse-grained data supplied through VCS snapshots.

Fluri et al. [15] proposed an algorithm to extract fine-grained changes from two snapshots of a source code file and implemented this algorithm in a tool, ChangeDistiller. ChangeDistiller represents the difference between two versions of a file as a sequence of atomic operations on the corresponding AST nodes. We also express changes as AST node operations, but our novel algorithm infers them directly from the fine-grained changes produced by a developer rather than from snapshots stored in VCS.

Kim et al. [29] proposed summarizing the structural changes between different versions of a source code file as high-level *change rules*. Change rules provide a cohesive description of related changes beyond deletion, addition, and removal of a textual element. Based on this idea, they created a tool that could automatically infer those change rules and present them as concise and understandable transformations to the programmer.

Weissgerber et al. [50] and Dig et al. [11] proposed tools for identifying refactorings between two different version of a source code. Such tools help developers gain better insights into the high-level transformations that occurred between different versions of a program.

All these tools detect structural changes in the evolving code using VCS snapshots. However, the results of our field study presented in Section 3 show that VCS snapshots provide incomplete and imprecise data, thus compromising the accuracy of these tools. The accuracy of such tools could be greatly improved by working on the fine-grained changes provided through a change-based software tool such as CodingTracker.

7.3 Tools for Fine-Grained Analysis of Code Evolution

Robbes et al. [42, 44] proposed to make a change the first-class citizen and capture it directly from an IDE as soon as it happens. They developed a tool, SpyWare [43], that implements these ideas. SpyWare gets notified by the Smalltalk compiler in the Squeak IDE whenever the AST of the underlying program changes. SpyWare records the captured AST modification events as operations on the corresponding AST nodes. Also, SpyWare records automated refactoring invocation events.

Although our work is inspired by similar ideas, our tool, CodingTracker, significantly differs from SpyWare. CodingTracker captures raw fine-grained code edits rather than a compiler's AST modification events. The recorded data is so precise that CodingTracker is able to replay it in order to reproduce the exact state of the evolving code at any point in time. Also, CodingTracker implements a novel AST node operations inferencing algorithm that does not expect the underlying code to be compilable or even fully parsable. Besides, CodingTracker captures a variety of evolution data that does not represent changes to the code, e.g., interactions with VCS, application and test runs, etc.

Sharon et al. [12] implemented EclipsEye, porting some ideas behind SpyWare to Eclipse IDE. Similarly to SpyWare, EclipsEye gets notified by Eclipse about AST changes in the edited application, but these notifications are limited to the high-level AST nodes starting from field and method declarations and up.

Omori and Maruyama [37, 38] developed a similar fine-grained operation recorder and replayer for the Eclipse IDE. In contrast to CodingTracker, their tool does not infer AST node operations but rather associates code edits with AST nodes, to which they might belong. Besides, CodingTracker captures more operations such as those that do not affect code like runs of programs and tests and version control system operations. The additional events that CodingTracker captures enabled us to study the test evolution patterns and the degree of loss of code evolution information in version control systems.

Yoon et al. [52] developed a tool, Fluorite, that records low-level events in Eclipse IDE. Fluorite captures sufficiently precise fine-grained data to reproduce the snapshots of the edited files. But the purpose of Fluorite is to study code editing patterns rather than software evolution in general. Therefore, Fluorite does not infer AST node operations from the collected raw data. Also, it does not capture such important evolution data as interactions with VCS, test runs, or effects of automated refactorings.

Chan et al. [6] proposed to conduct empirical studies on code evolution employing fine-grained revision history. They produce fine-grained revision history

of an application by capturing the snapshots of its files at every save and compilation action. Although such a history contains more detailed information about an application's code evolution than a common VCS, it still suffers from the limitations specific to snapshot-based approaches, in particular, the irregular intervals between the snapshots and the need to reconstruct the low level changes from the pairs of consecutive snapshots.

8 Conclusions

The primary source of data in code evolution research is the file-based Version Control System (VCS). Our results show that although popular among researchers, a file-based VCS provides data that is incomplete and imprecise. Moreover, many interesting research questions that involve code changes and other development activities (e.g., automated refactorings or test runs) require evolution data that is not captured by VCS at all.

We conducted a field study using CodingTracker, our Eclipse plug-in, that collects diverse evolution data. We analyzed the collected data and answered five code evolution research questions. We found that 37% of code changes are *shadowed* by other changes, and are not stored in VCS. Thus, VCS-based code evolution research is incomplete. Second, programmers intersperse different kinds of changes in the same commit. For example, 46% of refactored program entities are also edited in the same commit. This overlap makes the VCS-based research imprecise. The data collected by CodingTracker enabled us to answer research questions that could not be answered using VCS data alone. In particular, we discovered that 40% of test fixes involve changes to the tests, 24% of changes committed to VCS are untested, and 85% of changes to a method during an hour interval are clustered within 15 minutes.

These results confirm that more detailed data than what is stored in VCS is needed to study software evolution accurately.

Acknowledgments. This work was partially supported by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign, by the United States Department of Energy under Contract No. DE-F02-06ER25752, and by the National Science Foundation award number CCF 11-17960.

References

1. Adams, B., Jiang, Z.M., Hassan, A.E.: Identifying crosscutting concerns using historical code changes. In: ICSE (2010)
2. Apache Gump continuous integration tool, http://gump.apache.org/
3. Bamboo continuous integration and release management, http://www.atlassian.com/software/bamboo/
4. Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola Jr., J.J.: Code Bubbles: rethinking the user interface paradigm of integrated development environments. In: ICSE (2010)

5. Eclipse bug report, https://bugs.eclipse.org/bugs/show_bug.cgi?id=365233
6. Chan, J., Chu, A., Baniassad, E.: Supporting empirical studies by non-intrusive collection and visualization of fine-grained revision history. In: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (2007)
7. CVS - Concurrent Versions System, http://cvs.nongnu.org/
8. Daniel, B., Gvero, T., Marinov, D.: On test repair using symbolic execution. In: ISSTA (2010)
9. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: ReAssert: Suggesting repairs for broken unit tests. In: ASE (2009)
10. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: OOPSLA (2000)
11. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated Detection of Refactorings in Evolving Components. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 404–428. Springer, Heidelberg (2006)
12. EclipsEye, http://www.inf.usi.ch/faculty/lanza/Downloads/Shar07a.pdf
13. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. TSE 27, 1–12 (2001)
14. Eshkevari, L.M., Arnaoudova, V., Di Penta, M., Oliveto, R., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of identifier renamings. In: MSR (2011)
15. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. TSE 33, 725–743 (2007)
16. Gall, H., Hajek, K., Jazayeri, M.: Detection of logical coupling based on product release history. In: ICSM (1998)
17. Gall, H., Jazayeri, M., Klsch, R.R., Trausmuth, G.: Software evolution observations based on product release history. In: ICSM (1997)
18. Gall, H., Jazayeri, M., Krajewski, J.: CVS release history data for detecting logical couplings. In: IWMPSE (2003)
19. Girba, T., Ducasse, S., Lanza, M.: Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In: ICSM (2004)
20. Git - the fast version control system, http://git-scm.com/
21. Gorg, C., Weisgerber, P.: Detecting and visualizing refactorings from software archives. In: ICPC (2005)
22. Hassaine, S., Boughanmi, F., Guéhéneuc, Y.G., Hamel, S., Antoniol, G.: A seismology-inspired approach to study change propagation. In: ICSM (2011)
23. Hassan, A.E.: Predicting faults using the complexity of code changes. In: ICSE (2009)
24. Hindle, A., German, D.M., Holt, R.: What do large commits tell us?: a taxonomical study of large commits. In: MSR (2008)
25. Hudson extensive continuous integration server, http://hudson-ci.org/
26. Jenkins extendable open source continuous integration server, http://jenkins-ci.org/
27. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J. Softw. Maint. Evol. 19 (March 2007)
28. Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: ICSE (2011)
29. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: ICSE (2007)
30. Kim, S., James Whitehead Jr., E., Zhang, Y.: Classifying software changes: Clean or buggy? TSE 34(2) (2008)

31. Kim, S., Pan, K., Whitehead Jr., E.J.: Micro pattern evolution. In: MSR (2006)
32. Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J.: Automatic identification of bug-introducing changes. In: ASE (2006)
33. Lee, T., Nam, J., Han, D., Kim, S., In, H.P.: Micro interaction metrics for defect prediction. In: ESEC/FSE (2011)
34. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc. (1985)
35. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proc. IEEE 68(9), 1060–1076 (1980)
36. Mirzaaghaei, M., Pastore, F., Pezze, M.: Automatically repairing test cases for evolving method declarations. In: ICSM (2010)
37. Omori, T., Maruyama, K.: A change-aware development environment by recording editing operations of source code. In: MSR (2008)
38. Omori, T., Maruyama, K.: An editing-operation replayer with highlights supporting investigation of program modifications. In: IWMPSE-EVOL (2011)
39. Rahman, F., Posnett, D., Hindle, A., Barr, E., Devanbu, P.: BugCache for inspections: hit or miss? In: ESEC/FSE (2011)
40. Ratzinger, J., Sigmund, T., Vorburger, P., Gall, H.: Mining software evolution to predict refactoring. In: ESEM (2007)
41. Robbes, R.: Of Change and Software. Ph.D. thesis, University of Lugano (2008)
42. Robbes, R., Lanza, M.: A change-based approach to software evolution. ENTCS 166, 93–109 (2007)
43. Robbes, R., Lanza, M.: SpyWare: a change-aware development toolset. In: ICSE (2008)
44. Robbes, R., Lanza, M., Lungu, M.: An Approach to Software Evolution Based on Semantic Change. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 27–41. Springer, Heidelberg (2007)
45. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: MSR (2005)
46. Snipes, W., Robinson, B.P., Murphy-Hill, E.R.: Code hot spot: A tool for extraction and analysis of code change history. In: ICSM (2011)
47. Apache Subversion centralized version control, http://subversion.apache.org/
48. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: ICSE (2012)
49. Van Rysselberghe, F., Rieger, M., Demeyer, S.: Detecting move operations in versioning information. In: CSMR (2006)
50. Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE (2006)
51. Xing, Z., Stroulia, E.: Analyzing the evolutionary history of the logical design of object-oriented software. TSE 31, 850–868 (2005)
52. Yoon, Y., Myers, B.A.: Capturing and analyzing low-level events from the code editor. In: PLATEAU (2011)
53. Zimmermann, T., Nagappan, N., Zeller, A.: Predicting bugs from history. Software Evolution (2008)
54. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: ICSE (2004)

Evaluating the Design of the R Language

Objects and Functions for Data Analysis

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek

Purdue University

Abstract. R is a dynamic language for statistical computing that combines lazy functional features and object-oriented programming. This rather unlikely linguistic cocktail would probably never have been prepared by computer scientists, yet the language has become surprisingly popular. With millions of lines of R code available in repositories, we have an opportunity to evaluate the fundamental choices underlying the R language design. Using a combination of static and dynamic program analysis we assess the success of different language features.

1 Introduction

Over the last decade, the R project has become a key tool for implementing sophisticated data analysis algorithms in fields ranging from computational biology [7] to political science [11]. At the heart of the R project is a *dynamic, lazy, functional, object-oriented* programming language with a rather unusual combination of features. This computer language, commonly referred to as the R language [15,16] (or simply R), was designed in 1993 by Ross Ihaka and Robert Gentleman [10] as a successor to S [1]. The main differences with its predecessor, which had been designed at Bell labs by John Chambers, were the open source nature of the R project, vastly better performance, and, at the language level, lexical scoping borrowed from Scheme and garbage collection [1]. Released in 1995 under a GNU license, it rapidly became the lingua franca for statistical data analysis. Today, there are over 4 000 packages available from repositories such as CRAN and Bioconductor.[1] The R-forge web site lists 1 242 projects. With its 55 user groups, Smith [18] estimates that there are about 2 000 package developers, and over 2 million end users. Recent interest in the financial sector has spurred major companies to support R; for instance, Oracle is now bundling it as part of its Big Data Appliance product.[2]

As programming languages go, R comes equipped with a rather unlikely mix of features. In a nutshell, R is a dynamic language in the spirit of Scheme or JavaScript, but where the basic data type is the vector. It is functional in that functions are first-class values and arguments are passed by deep copy. Moreover, R uses lazy evaluation by default for all arguments, thus it has a pure functional core. Yet R does not optimize recursion, and instead encourages vectorized operations. Functions are lexically scoped and their local variables can be updated, allowing for an imperative programming style. R targets statistical computing, thus missing value support permeates all operations.

[1] `http://cran.r-project.org` and `http://www.bioconductor.org`

[2] `http://www.oracle.com/us/corporate/press/512001`

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 104–131, 2012.

The dynamic features of the language include forms of reflection over its environment, the ability to obtain source code for any unevaluated expression, and the `parse` and `eval` functions to dynamically treat text as code. Finally, the language supports objects. In fact, it has two distinct object systems: one based on single-dispatch generic functions, and the other on classes and multi-methods. Some surprising interactions between the functional and object parts of the language are that there is no aliasing, object structures are purely tree-shaped, and side effects are limited.

The R language can be viewed as a fascinating experiment in programming language design. Presented with a cornucopia of programming models, which one will users choose, and how? Clearly, any answer must be placed in the context of its problem domain: data analysis. How do these paradigms fit that problem domain? How do they strengthen each other and how do they interfere? Studying how these features are used in practice can yield insights for language designers and implementers. As luck would have it, the R community has several centralized code repositories where R packages are deposited together with test harnesses. Thus, not only do we have all the open source contributions made in the last 15 years, but we also have them in an executable format. This paper makes the following contributions:

- *Semantics of Core R:* Some of the challenges dealing with R come from the fact it is defined by a single implementation that exposes its inner workings through reflection. We make the first step towards addressing this issue. Combining a careful reading of the interpreter sources, the R manual [16], and extensive testing, we give the first formal account of the semantics of the core of the language. We believe that a precise definition of lazy evaluation in R was hitherto undocumented.
- *TraceR Framework:* We implemented TraceR, an open source framework for analysis of R programs. TraceR relies on instrumented interpreters and off-line analyzers along with static analysis tools.
- *Corpus Gathering:* We curated a large corpus of R programs composed of over 1 000 executable R packages from the Bioconductor and CRAN repositories, as well as hand picked end-user codes and small performance benchmark programs that we wrote ourselves.
- *Implementation Evaluation:* We evaluate the status of the R implementation. While its speed is not acceptable for use in production systems, many end users report being vastly more productive in R than in other languages. R is decidedly single-threaded, its semantics has no provisions for concurrency, and its implementation is hopelessly non-thread safe. Memory usage is also an issue; even small programs have been shown to use immoderate amounts of heap for data and meta-data. Improving speed and memory usage will require radical changes to the implementation, and a tightening of the language definition.
- *Language Evaluation:* We examine the usage and adoption of different language features. R permits many programming styles, access to implementation details, and little enforcement of data encapsulation. Given the large corpus at hand, we look at the usage impacts of these design decisions.

The code and data of our project are available in open source from:
`http://r.cs.purdue.edu/`

2 An R Primer

We introduce the main concepts of the R programming language. To understand the design of R, it is helpful to consider the end-user experience that the designers of R and S were looking for. Most sessions are interactive, the user loads data into the virtual machine and starts by plotting the data and making various simple summaries. If those do not suffice, there are some 4 338 statistical packages that can be applied to the data. Programming proper only begins if the modeling steps become overly repetitive, in which case they can be packaged into simple top-level functions. It is only if the existing packages do not precisely match the user's needs that a new package will be developed. The design of the language was thus driven by the desire to be intuitive, so users who only require simple plotting and summarization can get ahead quickly. For package developers, the goal was flexibility and extendibility. A tribute to the success of their approach is the speed at which new users can pick up R; in many statistics departments the language is introduced in a week.

The basic data type in R is the vector, an ordered collection of values of the same kind. These values can be either numerics, characters, or logicals. Other data types include lists (i.e., heterogeneous vectors) and functions. Matrices, data frames, and objects are built up from vectors. A command for creating a vector and binding it to `x` is:

```
x <- c(1, 2.1, 3, NA)
```

Missing values, `NA`, are crucial for statistical modeling and impact the implementation, as they must be represented and handled efficiently. Arithmetic operations on vectors are performed element by element, and shorter vectors are automatically extended to match longer ones by an implicit extension process. For instance,

```
v <- 1 + 3*x
```

binds the result of the expression `1+3*x` to `v`. There are three different vectors: `x` and two vectors of length one, namely the numeric constants `1` and `3`. To evaluate the expression, R will logically extend the constant vectors to match the length of `x`. The result will be a new vector equivalent to `c(4,7.3,10,NA)`. Indexing operators include:

```
v1 <- x[1:3];      v2 <- x[-1];      x[is.na(x)] <- 0
```

here `v1` is bound to a new vector composed of the first three elements of `x`, `v2` is bound to a new vector with everything but the first value of `x`, and finally, `x` is updated with 0 replacing any missing values.

In R, computation happens by evaluating functions. Even the assignment, `x<-1`, is a call to the built-in `assign("x",1)`. This design goes as far as making the `(` in a parenthesized expression a function call. Functions are first class values that can be created, stored and invoked. So,

```
pow <- function(b,e=2) if(e==1) b else b*pow(b,e-1)
```

creates a function which takes two arguments and binds it to `pow`. Function calls can specify parameters either by position or by name. Parameters that have default values, such as `e` above, need not be passed. Thus, there are three equivalent ways to call `pow`:

```
pow(3);      pow(3,2);      pow(e=2,3)
```

The calls all return the 1-element vector `9`. Named arguments are significantly used; functions such as `plot` accept over 20 arguments. The language also supports '`...`' in function definition and calls to represent a variable number of values. Explicit type declarations are not required for variables and functions. True to its dynamic roots, R checks type compatibility at runtime, performing conversions when possible to provide best-effort semantics and decrease errors.

R is lazy, thus evaluation of function arguments is delayed. For example, the `with` function can be invoked as follows:

```
with(formaldehyde, carb*optden)
```

Its semantics is similar to the JavaScript `with` statement. The second argument is evaluated in the context of the first which must be an environment (a list or a special kind of vector). This behavior relies on lazy evaluation, as it is possible that neither `carb` or `optden` are defined at the point of call. Arguments are boxed into *promises* which contain the expression passed as an argument and a reference to the current environment. Promises are evaluated transparently when their value is required. The astute reader will have noticed that the above example clashes with our claim that R is lexically scoped. As is often the case, R is lexically scoped up to the point it is not. R is above all a dynamic language with full reflective access to the running program's data and representation. In the above example, the implementation of `with` sidesteps lexical scoping by reflectively manipulating the environment. This is done by a combination of lazy evaluation, dynamic name lookup, and the ability turn code into text and back:

```
with.default <- function(env, expr, ...)
    eval(substitute(expr),env, enclose=parent.frame())
```

The function uses `substitute` to retrieve the unevaluated parse tree of its second argument, then evaluates it with `eval` in the environment constituted by composing the first argument with the lexically enclosing environment. The '`...`' is used to discard any additional arguments.

R associates attributes to all data structures, thus every vector, list, or function has a hidden map that associates symbols to values. For a vector, these include length, dimensions, and column names. Attributes are a key to R's extensibility. They are used as hooks for many purposes. As we will see in the next section, the object system relies on attributes to encode class membership. It is sometimes the case that all the interesting data for some value is squirreled away in attributes. Attributes can be updated by an assignment, e.g., to turn the vector `x` into a 2-by-2 matrix:

```
attr(x, "dim") <- c(2,2)
```

R has two different object systems. The simplest one uses a `class` attribute for implementing ad-hoc polymorphism. This attribute holds a series of strings denoting base class and parent classes in order. Any data structure can be labeled as the programmer wishes. The new object system allows for true class definitions and instance creation. It gives the programmer a similar multiple inheritance model as the early object system, but now allows for virtual classes and multi-method dispatch.

3 The Three Faces of R

We now turn to R's support for different paradigms.

3.1 Functional

R has a functional core reminiscent of Scheme and Haskell.

Functions as first-class objects. Functional languages manipulate functions as first-class objects. Functions can be created and bound to symbols. They can be passed as arguments, or even given alternate names such as `f<-factorize; f(v)`.

Scoping. Names are lexically scoped with no difference between variable and function declarations, as in Scheme [6]. All symbols introduced during the evaluation of a function are collected in a *frame*. Frames are linked to their lexically enclosing frame to compose environments. However, unlike many languages, bindings are performed dynamically [8]. Symbols can be added to an environment after it has been entered. This forces name resolution to be performed during function evaluation. The mechanism is subtle and has led to mistaken claims that R is not lexically scoped[3]. Consider,

```
function(){ f<-function()x;     x<-42;     f() }
```

where the function bound to `f` accesses a variable, `x`, that does not exist at the time of definition. But when `f` is called, lookup will succeed as `x` is in scope. Scoping has another somewhat surprising wrinkle, lookup is context sensitive. Looking up `c` and `c()` can yield different results from CommonLisp [19] or Scheme. By default, c is bound to the built-in function that creates new vectors. In these examples, the first code fragment binds 42 to `c`. When `c` is looked up in the function call, the binding is skipped because 42 is not a function, and lookup instead returns the default definition of `c` which is indeed a function. The second code fragment adds an assignment of `c` to `d`. When `d` is looked up, the only definition of `d` in the environment is the one that binds it to 42, so the call fails.[4] This is an example of best effort semantics; the lookup rules try to find a function when in a call context, which can yield surprising results.

```
c <- 42
c(2,3)
```

```
c <- 42
d <- c
d(2,3)
```

Lazy. R does not evaluate function arguments at calls. Instead, expressions are boxed into promises that capture the lexical environment. Promises are *forced* whenever their value is required by the computation. Languages like Haskell [9] delay evaluation as much as possible. This allows, e.g., for the elegant definition of infinite data structures. The R manual [16] does not state when promises are forced. As R is not purely functional, the order of evaluation is observable through side effects performed or observed by promises. Our investigation uncovered that promises are evaluated aggressively. They typically do not outlive the function that they are passed into.[5] Another surprising discovery is that name lookup will force promises in order to determine if a symbol is bound to a function. Consider the following three-argument function declaration:

```
function(y, f, z) { f(); return( y ) }
```

[3] E. Blair, A critique of R (2004) `http://fluff.info/blog/arch/00000041.htm`

[4] Our core R semantics models this behavior with the [FORCEF] and [GETF] rules of Fig. 3.

[5] Unbounded data structures can be created by hiding promises in closures, e.g., `Cons <-function(x,y)list(function()x, function()y)`. An application is needed to get the value, e.g., `Car <-function(cons)cons[[1]]()`.

If it is not a function, evaluation forces `f`. In fact, each definition of `f` in the lexical environment is forced until a function is found. `y` is forced as well and `z` is unevaluated.

Referential transparency. A language is referentially transparent if any expression can be replaced by its result. This holds if evaluating the expression does not side effect other values in the environment. In R, all function arguments are passed by value, thus all updates performed by the function are visible only to that function. On the other hand, environments are mutable and R provides the super assignment operator (`<<-`) in addition to its local one (`<-`). Local assignment, `x<-1`, either introduces a new symbol or updates the value of an existing symbol in the current frame. This side effect remains local and is arguably easier to reason about than generalized side effects. The super assignment operator is a worse offender as it skips the current frame and operates on the rest of the environment. So, `x<<-1`, will ignore any definition of `x` in the current frame and update the first existing `x` anywhere in the environment or, if not found, define a new binding for `x` at the top-level. This form of side effect is harder to reason about as it is non-local and may be observed directly by other functions.

Parameters. R gives programmers much freedom in how functions are defined and called. Function declarations can specify default values and a variable number of parameters. Function calls can pass parameters by position or by name, and omit parameters with default values. Consider the `my` function declaration, it has three parameters, `a`, `k`, and `p`. The ellipsis specifies a variable number of parameters which can be accessed by the array notation or passed on to another function in bulk. This function can be called in different ways, for instance,

```
my <-
  function(a,k=min(d),...,p=TRUE){
    a <- as.matrix(a);
    d <- dim(a);
    l <- k+1;
    if (p) # some behavior.
}
```

```
my(x);    my(x, y);    my(x, y, z);    my(k=y, x, z, p=FALSE)
```

A valid call must have at least one, positional, parameter. Any argument with a default value may be omitted. Any argument may be passed by name, in which case the order in which it appears is irrelevant. Arguments occurring after an ellipsis must be passed by name. The default values of arguments can be expressions; they are boxed into promises and evaluated within the same environment as the body of the function. This means that they can use internal variables in the function's body. So `min(d)` above refers to `d`, which is only created during evaluation of the function. As `k` is always forced after `d` has been defined, the function will work as intended. But this shows that code can be sensitive to the order of evaluation, which can lead to subtle bugs.

3.2 Dynamic

Given R's interactive usage, dynamic features are natural. These features are intended to increase the expressiveness and flexibility of the language, but complicate the implementor's task.

Dynamic typing. R is dynamically typed. Values have types, but variables do not. This dynamic behavior extends to variable growth and casting. For instance:

```
v <- TRUE;    v[2] <- 1;    v[4] <- "1"
```

Vector `v` starts as a logical vector of length one, then `v` grows and is cast by the assignment to a numerical vector of length two, equivalent to `c(1, 1)`. The last assignment turns `v` into a string vector of length four, equivalent to `c("1", "1", NA, "1")`.

Dynamic evaluation. R allows code to be dynamically evaluated through the `eval` function. Unevaluated expressions can be created from text with the `quote` function, and variable substitution (without evaluation) can be done with `substitute` and partial substitution with `bquote`. Further, expressions can be reduced back to input strings with the `substitute` and `deparse` functions. The R manual [16] mentions these functions as useful for dynamic generation of chart labels, but they are used for much more.

Extending the language. One of the motivations to use lazy evaluation in R is to extend the language without needing macros. But promises are only evaluated once, so implementing constructs like a `while` loop, which must repeatedly evaluate its body and guard, takes some work. The `substitute` function can be used to get the source text of a promise, the expression that was passed into the function, and `eval` to execute it. Consider this implementation of a while loop in user code,

```
mywhile <- function(cond, body)
   repeat if(!eval.parent(substitute(cond))) break
             else eval.parent(substitute(body))
```

Not all language extensions require reflection, lazy evaluation can be sufficient. The implementation of `tryCatch` is roughly,

```
tryCatch <- function(expr, ...) {
  # set up handlers specified in ...
  expr
}
```

Explicit Environment Manipulation. Beyond these common dynamic features, R's reflective capabilities also expose its implementation to the user. Environments exist as a data type. Users can create new environments, and view or modify existing ones, including those used in the current call stack. Closures can have their parameters, body, or environment changed after they are defined. The call stack is also open for examination at any time. There are ways to access any closure or frame on the stack, or even return the entire stack as a list. With this rich set of building blocks, user code can implement whatever scoping rules they like.

3.3 Object Oriented

R's use of objects comes from S. Around 1990, a `class` attribute was added to S3 for ad-hoc polymorphism [4]. The value of this attribute was simply a list of strings used to dispatch methods. In 1998, S4 added proper classes and multi-methods [3].

S3. In the S3 object model there are neither class entities nor prototypes. Objects are normal R values tagged by an attribute named `class` whose value is a vector of strings. The strings represent the classes to which the object belongs. Like all attributes, an object's class can be updated at any time. As no structure is required on objects, two instances that

have the same class tag may be completely different. The only semantics for the values in the `class` attribute is to specify an order of resolution for methods. Methods in S3 are functions which, in their body, call the dispatch function `UseMethod`. The dispatch function takes a string `name` as argument and will perform dispatch by looking for a function `name.cl` where `cl` is one of the values in the object's `class` attribute. Thus a call to `who(me)` will access the `me` object's class attribute and perform a lookup for each class name until a match is found. If none is found, the function `who.default` is called. This mechanism is complemented by `NextMethod` which calls the *super* method according to the Java terminology. The function follows the same algorithm as `UseMethod`, but starts from the successor of the name used to find the current function. Consider the following example which defines a generic method, `who`, with implementations for class the `man` as well as the `default` case, and creates an object of class `man`. Notice that the vector `me` dynamically acquires class `man` as a side effect. `UseMethod` may take multiple arguments, but dispatches only on the first one.

```
who <-         function(x) UseMethod("who")
who.man <-     function(x) print("Ceasar!")
who.default <- function(x) print("??")
me <- 42;           who(me)    # prints "??"
class(me) <- 'man';who(me)     # prints "Ceasar!"
```

S4. The S4 object model, reminiscent of CLOS [12], adds a class-based system to R. A class is defined by a call to `setClass` with an ordered list of parent classes and a representation. Multiple inheritance is supported, and repeated inheritance is allowed, but only affects the method's dispatch algorithm. A class' representation describes the fields, or slots, introduced by that class. Even though R is dynamically typed, slots must be assigned a class name. Slots can be redefined covariantly, i.e., a slot redefinition can be a subclass of the class tag used in the previous declaration. When a class inherits a slot with the same name from two different paths, the class tag coming from the first superclass is retained, and tags from other parents must be subclasses of that tag. Classes can be redeclared at any time. When a class is modified, existing instances of the class retain their earlier definition. Redeclaration can be prevented by the `sealClass` function. Objects are instantiated by calling `new`. A prototype object is created with the arguments to `new` and passed along to the `initialize` generic function which copies the prototype into the new object's slots. This prototype is only a template and does not contain any methods. Any values left unset become either NA, or a zero-length vector. Classes without a representation, or classes with `VIRTUAL` in their representation, are abstract classes and cannot be instantiated. Another mechanism for changing the behavior of existing classes is to define *class unions*. A class union introduces a new virtual class that is the parent of a list of existing classes. The main role of class union is to change the result of method dispatch for existing classes. The following example code fragment defines a colored point class, creates an instance of the class, and reads its `color` slot.

```
setClass("Point", representation(x="numeric", y="numeric"))
setClass("Color", representation(color="character"))
setClass("CP", contains=c("Point","Color"))
l <- new("Point", x = 1, y = 1)
r <- new("CP", x = 0, y = 0, color = "red")
r@color
```

Methods are introduced at any point outside of any class by a call to `setGeneric`. Individual method bodies for some classes are then defined by `setMethod`. R supports multi-methods [2], i.e., dispatch considers the classes of multiple arguments to determine which function to call. Multi-methods make it trivial to implement binary methods, they obviate the need for the visitor pattern or other forms of double dispatch, and reduce the number of explicit subclass tests in users' code. The following defines an `add` method that will operate differently on points and colored points:

```
setGeneric("add", function(a, b) standardGeneric("add"))
setMethod("add", signature("Point", "Point"),
  function(a, b) new("Point", x= a@x+b@x, y=a@y+b@y))
setMethod("add", signature("CP", "CP"),
  function(a, b) new("CP",x=a@x+b@x,y=a@y+b@y,color=a@color)
```

R does not prevent the declaration of ambiguous multi-methods. At each method call, R will attempt to find the best match between the classes of the parameters and the signatures of method bodies. Thus `add(r, l)` would be treated as the addition of two "Point" objects. The resolution algorithm differs from Clos's and if more than one method is applicable, R will pick one and emit a warning. One unfortunate side effect of combining generic functions and lazy evaluation is that method dispatch forces promises to assess the class of each argument. Thus when S4 objects are used, evaluation of arguments becomes strict.

4 A Semantics for Core R

This section gives a precise semantics to the core of the R language. To the best of our knowledge this is the first formal definition of key concepts of the language. The semantics was derived from test cases and inspection of the source code of version 2.12 of the R interpreter. Core R is a proper subset of the R language. Any expression in Core R behaves identically in the full language. Some features are not covered for brevity: logicals and complex numbers, partial keywords, variadic argument lists, dot-dot symbols, superfluous arguments, generalized array indexing and subsetting. Generalized assignment, `f(x)<-y`, requires small changes to be properly supported, essentially desugaring to function calls such as `'f<-'(x,y)` and additional assignments. Perhaps the most glaring simplification is that we left out reflective operation such as `eval` and `substitute`. As the object system is built on those, we will only hint at its definition.

$$
\begin{array}{rcl}
\mathtt{e} & ::= & \mathtt{n} \mid \mathtt{s} \mid \mathtt{x} \mid \mathtt{x}[[\mathtt{e}]] \mid \{\mathtt{e};\mathtt{e}\} \\
& \mid & \mathtt{function}(\overline{\mathtt{f}})\,\mathtt{e} \\
& \mid & \mathtt{x}(\overline{\mathtt{a}}) \mid \mathtt{x} <\!\!- \mathtt{e} \mid \mathtt{x} <\!\!<\!\!- \mathtt{e} \\
& \mid & \mathtt{x}[[\mathtt{e}]] <\!\!- \mathtt{e} \mid \mathtt{x}[[\mathtt{e}]] <\!\!<\!\!- \mathtt{e} \\
& \mid & \mathtt{attr}(\mathtt{e},\mathtt{e}) \mid \mathtt{attr}(\mathtt{e},\mathtt{e}) <\!\!- \mathtt{e} \\
& \mid & u \mid \nu(\overline{\mathtt{a}}) \\
\mathtt{f} & ::= & \mathtt{x} \mid \mathtt{x} = \mathtt{e} \\
\mathtt{a} & ::= & \mathtt{e} \mid \mathtt{x} = \mathtt{e}
\end{array}
$$

Fig. 1. Syntax

The syntax of Core R, shown in Fig. 1, consists of expressions, denoted by e, ranging over numeric literals, string literals, symbols, array accesses, blocks, function declarations, function calls, variable assignments, variable super-assignments, array assignments, array super-assignments, and attribute extraction and assignment. Expressions

also include values, u, and partially reduced function calls, $\nu(\overline{\mathtt{a}})$, which are not used in the surface syntax of the language but are needed during evaluation. The parameters of a function declaration, denoted by `f`, can be either variables or variables with a default value, an expression `e`. Symmetrical arguments of calls, denoted `a`, are expressions which may be named by a symbol. We use the notation $\overline{\mathtt{a}}$ to denote the possibly empty sequence $\mathtt{a}_1 \ldots \mathtt{a}_n$. Programs compute over a heap, denoted H, and a stack, S, as shown in Fig. 2. For simplicity, the heap differentiates between three kinds of addresses: frames, ι, promises, δ, and data objects, ν. The notation $H[\iota/F]$ denotes the heap H extended with a mapping from ι to F. The metavariable $\nu_\perp$ denotes ν extended with the distinguished reference $\perp$ which is used for missing values. Metavariable α ranges over pairs of possibly missing addresses, $\nu_\perp\,\nu'_\perp$. The metavariable u ranges over both promises and data references. Data objects, κ^α, consist of a primitive value κ and attributes α. Primitive values can be either an array of numerics, $\mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_n]$, an array of strings, $\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_n]$, an array of references $\mathtt{gen}[\nu_1 \ldots \nu_n]$, or a function, $\lambda\overline{\mathtt{f}}.\mathtt{e}, \Gamma$, where Γ is the function's environment. A frame, F, is a mapping from a symbol to a promise or data reference. An environment, Γ, is a sequence of frame references. Finally, a stack, S, is a sequence of pairs, $\mathtt{e}\;\Gamma$, such that `e` is the current expression and Γ is the current environment.

$$
\begin{array}{l}
H ::= \emptyset \mid H[\iota/F] \\
\quad \mid H[\delta/\mathtt{e}\;\Gamma] \mid H[\delta/\nu] \\
\quad \mid H[\nu/\kappa^\alpha] \\
\alpha ::= \nu_\perp\,\nu_\perp \quad u ::= \delta \mid \nu \\
\kappa ::= \mathtt{num}[\overline{\mathtt{n}}] \mid \mathtt{str}[\overline{\mathtt{s}}] \\
\quad \mid \mathtt{gen}[\overline{\nu}] \mid \lambda\overline{\mathtt{f}}.\mathtt{e}, \Gamma \\
F ::= [] \mid F[\mathtt{x}/u] \\
\Gamma ::= [] \mid \iota * \Gamma \\
S ::= [] \mid \mathtt{e}\;\Gamma * S
\end{array}
$$

Fig. 2. Data

Reduction relation. The semantics of Core R is defined by a small step operational semantics with evaluation contexts [21]. The reduction relation $S;H \Longrightarrow S';H'$, shown in Fig. 3, takes a stack S and a heap H and performs one step of reduction. The rules

$$
\frac{\mathtt{e}\;\Gamma; H \;\rightarrow\; \mathtt{e}'; H'}{\mathbb{C}[\mathtt{e}]\;\Gamma * S; H \;\Longrightarrow\; \mathbb{C}[\mathtt{e}']\;\Gamma * S; H'}\;[\textsc{Exp}]
\qquad
\frac{H(\delta) = \mathtt{e}\;\Gamma'}{\mathbb{C}[\delta]\;\Gamma * S; H \;\Longrightarrow\; \mathtt{e}\;\Gamma' * \mathbb{C}[\delta]\;\Gamma * S; H}\;[\textsc{ForceP}]
$$

$$
\frac{\mathsf{getfun}(H, \Gamma, \mathtt{x}) = \delta}{\mathbb{C}[\mathtt{x}(\overline{\mathtt{a}})]\;\Gamma * S; H \;\Longrightarrow\; \delta\;\Gamma * \mathbb{C}[\mathtt{x}(\overline{\mathtt{a}})]\;\Gamma * S; H}\;[\textsc{ForceF}]
\qquad
\frac{\mathsf{getfun}(H, \Gamma, \mathtt{x}) = \nu}{\mathbb{C}[\mathtt{x}(\overline{\mathtt{a}})]\;\Gamma * S; H \;\Longrightarrow\; \mathbb{C}[\nu(\overline{\mathtt{a}})]\;\Gamma * S; H}\;[\textsc{GetF}]
$$

$$
\frac{H(\nu) = \lambda\overline{\mathtt{f}}.\mathtt{e}, \Gamma' \quad \mathsf{args}(\overline{\mathtt{f}}, \overline{\mathtt{a}}, \Gamma, \Gamma', H) = F, \Gamma'', H'}{\mathbb{C}[\nu(\overline{\mathtt{a}})]\;\Gamma * S; H \;\Longrightarrow\; \mathtt{e}\;\Gamma'' * \mathbb{C}[\nu(\overline{\mathtt{a}})]\;\Gamma * S; H'}\;[\textsc{InvF}]
$$

$$
\frac{H' = H[\delta/\nu]}{\mathbb{R}[\nu]\;\Gamma' * \mathbb{C}[\delta]\;\Gamma * S; H \;\Longrightarrow\; \mathbb{C}[\delta]\;\Gamma * S; H'}\;[\textsc{RetP}]
\qquad
\frac{}{\mathbb{R}[\nu]\;\Gamma' * \mathbb{C}[\nu'(\overline{\mathtt{a}})]\;\Gamma * S; H \;\Longrightarrow\; \mathbb{C}[\nu]\;\Gamma * S; H'}\;[\textsc{RetF}]
$$

Evaluation Contexts:

$$
\begin{array}{l}
\mathbb{C} ::= [] \mid \mathtt{x} <\!\!- \mathbb{C} \mid \mathtt{x}[[\mathbb{C}]] \mid \mathtt{x}[[\mathtt{e}]] <\!\!- \mathbb{C} \mid \mathtt{x}[[\mathbb{C}]] <\!\!- \nu \mid \{\mathbb{C}; \mathtt{e}\} \mid \{\nu; \mathbb{C}\} \\
\quad \mid \mathtt{attr}(\mathbb{C}, \mathtt{e}) \mid \mathtt{attr}(\nu, \mathbb{C}) \mid \mathtt{attr}(\mathtt{e}, \mathtt{e}) <\!\!- \mathbb{C} \mid \mathtt{attr}(\mathbb{C}, \mathtt{e}) <\!\!- \nu \mid \mathtt{attr}(\nu, \mathbb{C}) <\!\!- \nu \\
\mathbb{R} ::= [] \mid \{\nu; \mathbb{R}\}
\end{array}
$$

Fig. 3. Reduction relation $\Longrightarrow$

rely on two evaluation contexts, $\mathbb{C}$, to return the next expression to evaluate and $\mathbb{R}$, to return the result of a sequence of expressions. There are seven reduction rules. Rule [EXP] deals with expressions, where $\mathbb{C}[\mathtt{e}]$ uniquely identifies the next expression $\mathtt{e}$ to evaluate. The expression is reduced in a single step, $\mathtt{e}\,\Gamma; H \rightarrow \mathtt{e}'; H'$, where $\mathtt{e}'$ is resulting expression. H' is the modified heap. If the expression is a promise, $\mathbb{C}[\delta]$, and δ has not been evaluated, rule [FORCEP] will push a new frame on the stack containing the body of the promise, $\mathtt{e}\,\delta * \Gamma'$. Rule [RETP] pops a fully evaluated promise frame and binds the result to a promise address. Context sensitive lookup is implemented by [FORCEF] and [GETF]. The former forces the evaluation of promises bound to the name of the function being looked up, the latter selects a reference, ν, to a function. The $\mathsf{getfun}()$ auxiliary function, defined in Fig. 4, looks up $\mathtt{x}$ in the environment, skipping over bindings to data objects. Function invocation is handled by [INVF], which retrieves the function bound to ν and invokes $\mathsf{args}()$ to process the arguments $\overline{\mathtt{a}}$ and the default values $\overline{\mathtt{f}}$ of the call. The output of $\mathsf{args}()$ is a mapping from parameters to values, F, an environment, Γ'', and a modified heap, H'. For each argument, a promise is allocated in the heap and the current environment is captured. The rule [RETF] simply pops the evaluated frame and replaces the call with its result.

The $\rightarrow$ relation has fourteen rules dealing with expressions, shown in Fig. 5, along with some auxiliary definitions given in Fig. 18 (where s and g denote functions that convert the type of their argument to a string and vector respectively). The first two rules deal with numeric and string literals. They simply allocate a vector of length one of the

$$\frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \nu \quad H(\nu) = \lambda\overline{\mathtt{f}}.\mathtt{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \nu}\ [\textsc{GetF1}] \qquad \frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \nu \quad H(\nu) \neq \lambda\overline{\mathtt{f}}.\mathtt{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \mathsf{getfun}(H,\Gamma',\mathtt{x})}\ [\textsc{GetF2}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \delta \quad H(\delta) = \nu \quad H(\nu) = \lambda\overline{\mathtt{f}}.\mathtt{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \nu}\ [\textsc{GetF3}] \qquad \frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \delta \quad H(\delta) = \mathtt{e}\,\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \delta}\ [\textsc{GetF4}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \delta \quad H(\delta) = \nu \quad H(\nu) \neq \lambda\overline{\mathtt{f}}.\mathtt{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \mathsf{getfun}(H,\Gamma',\mathtt{x})}\ [\textsc{GetF5}]$$

$$\frac{\mathsf{split}(\overline{\mathtt{a}},P,N) = P',N'}{\mathsf{split}(\mathtt{x} = \mathtt{e}\,\overline{\mathtt{a}},P,N) = P',\mathtt{x} = \mathtt{e}N'}\ [\textsc{Split1}] \qquad \frac{\mathsf{split}(\overline{\mathtt{a}},P,N) = P',N'}{\mathsf{split}(\mathtt{e}\,\overline{\mathtt{a}},P,N) = \mathtt{e}P',N'}\ [\textsc{Split2}] \qquad \frac{}{\mathsf{split}([],P,N) = P,N}\ [\textsc{Split3}]$$

$$\frac{\mathsf{split}(\overline{\mathtt{a}},[],[]) = P,N \quad \iota\,\mathtt{fresh} \quad \Gamma'' = \iota * \Gamma' \quad \mathsf{args2}(\overline{\mathtt{f}},P,N,\Gamma,\Gamma'',H) = F,H' \quad H'' = H'[\iota/F]}{\mathsf{args}(\overline{\mathtt{f}},\overline{\mathtt{a}},\Gamma,\Gamma',H) = F,\Gamma'',H''}\ [\textsc{Args}]$$

$$\frac{(\mathtt{f}_0 \equiv \mathtt{x} \vee \mathtt{f}_0 \equiv \mathtt{x} = \mathtt{e}') \quad N \equiv N'\mathtt{x} = \mathtt{e}N'' \quad \mathsf{args2}(\overline{\mathtt{f}},P,N'N'',\Gamma,\Gamma',H) = F,H' \quad \delta\,\mathtt{fresh} \quad H'' = H'[\delta/\mathtt{e}\,\Gamma]}{\mathsf{args2}(\mathtt{f}_0\overline{\mathtt{f}},P,N,\Gamma,\Gamma',H) = F[\mathtt{x}/\delta],H''}\ [\textsc{Args1}]$$

$$\frac{(\mathtt{f}_0 \equiv \mathtt{x} \vee \mathtt{f}_0 \equiv \mathtt{x} = \mathtt{e}') \quad \mathtt{x} \notin N \quad \mathsf{args2}(\overline{\mathtt{f}},P,N,\Gamma,\Gamma',H) = F,H' \quad \delta\,\mathtt{fresh} \quad H'' = H'[\delta/\mathtt{e}\,\Gamma]}{\mathsf{args2}(\mathtt{f}_0\overline{\mathtt{f}},\mathtt{e}P,N,\Gamma,\Gamma',H) = F[\mathtt{x}/\delta],H''}\ [\textsc{Args2}]$$

$$\frac{\mathtt{x} \notin N \quad \mathsf{args2}(\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F,H'}{\mathsf{args2}(\mathtt{x}\,\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F[\mathtt{x}/\bot],H'}\ [\textsc{Args3}]$$

$$\frac{\mathtt{x} \notin N \quad \mathsf{args2}(\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F,H' \quad \delta\,\mathtt{fresh} \quad H'' = H'[\delta/\mathtt{e}\,\Gamma']}{\mathsf{args2}(\mathtt{x} = \mathtt{e}\,\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F[\mathtt{x}/\delta],H''}\ [\textsc{Args4}]$$

$$\frac{}{\mathsf{args2}([],[],[],\Gamma,\Gamma',H) = [],H}\ [\textsc{Args5}]$$

Fig. 4. Auxiliary definitions: Function lookup and argument processing

$$\frac{\nu\,\texttt{fresh} \quad \alpha = \bot\,\bot \quad H' = H[\nu/\texttt{num}[\texttt{n}]^{\alpha}]}{\texttt{n}\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{Num}] \qquad \frac{\nu\,\texttt{fresh} \quad \alpha = \bot\,\bot \quad H' = H[\nu/\texttt{str}[\texttt{s}]^{\alpha}]}{\texttt{s}\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{Str}]$$

$$\frac{\nu\,\texttt{fresh} \quad \alpha = \bot\,\bot \quad H' = H[\nu/\lambda\overline{\texttt{f}}.\texttt{e}, \Gamma^{\alpha}]}{\texttt{function}(\overline{\texttt{f}})\;\texttt{e}\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{Fun}]$$

$$\frac{\Gamma(H, \texttt{x}) = u}{\texttt{x}\,\Gamma; H \;\rightarrow\; u; H}\;[\textsc{Find}] \qquad \frac{H(\delta) = \nu}{\delta\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{GetP}]$$

$$\frac{\mathsf{cpy}(H, \nu) = H', \nu' \quad \Gamma = \iota * \Gamma' \quad H(\iota) = F \quad F' = F[\texttt{x}/\nu'] \quad H'' = H'[\iota/F']}{\texttt{x} <\!\!- \nu\,\Gamma; H \;\rightarrow\; \nu; H''}\;[\textsc{Ass}]$$

$$\frac{\mathsf{cpy}(H, \nu) = H', \nu' \quad \Gamma = \iota * \Gamma' \quad \mathsf{assign}(\texttt{x}, \nu', \Gamma', H') = H''}{\texttt{x} <\!\!<\!\!- \nu\,\Gamma; H \;\rightarrow\; \nu; H''}\;[\textsc{DAss}]$$

$$\frac{\Gamma(H, \texttt{x}) = \nu' \quad \mathsf{readn}(\nu, H) = \texttt{m} \quad \mathsf{get}(\nu', \texttt{m}, H) = \nu'', H'}{\texttt{x}[[\nu]]\,\Gamma; H \;\rightarrow\; \nu''; H'}\;[\textsc{Get}]$$

$$\frac{\mathsf{cpy}(H, \nu') = H', \nu'' \quad \Gamma = \iota * \Gamma' \quad \iota(H', \texttt{x}) = \nu''' \quad \mathsf{readn}(\nu, H') = \texttt{m} \quad \mathsf{set}(\nu''', \texttt{m}, \nu'', H') = H''}{\texttt{x}[[\nu]] <\!\!- \nu'\,\Gamma; H \;\rightarrow\; \nu'; H''}\;[\textsc{SetL}]$$

$$\frac{\mathsf{cpy}(H, \nu') = H', \nu'' \quad \Gamma = \iota * \Gamma' \quad H'(\iota) = F \quad \texttt{x} \notin F \quad \Gamma'(H', \texttt{x}) = \nu''' \quad \mathsf{cpy}(H', \nu''') = H'', \nu'''' \quad F' = F[\texttt{x}/\nu''''] \quad H''' = H''[\iota/F'] \quad \mathsf{readn}(\nu, H) = \texttt{m} \quad \mathsf{set}(\nu'''', \texttt{m}, \nu'', H''') = H''''}{\texttt{x}[[\nu]] <\!\!- \nu'\,\Gamma; H \;\rightarrow\; \nu'; H''''}\;[\textsc{SetG}]$$

$$\frac{H(\nu) = \kappa^{\alpha} \quad \alpha = \nu_{\bot}\,\nu'_{\bot} \quad \mathsf{index}(\nu', \nu'_{\bot}, H) = \texttt{n} \quad \mathsf{get}(\nu_{\bot}, \texttt{n}, H) = \nu''}{\texttt{attr}(\nu, \nu')\,\Gamma; H \;\rightarrow\; \nu''; H}\;[\textsc{GetA}]$$

$$\frac{H(\nu) = \kappa^{\alpha} \quad \alpha = \nu_{\bot}\,\nu'_{\bot} \quad \mathsf{index}(\nu', \nu'_{\bot}, H) = \texttt{n} \quad \mathsf{set}(\nu, \texttt{n}, \nu'', H) = H'}{\texttt{attr}(\nu, \nu') <\!\!- \nu''\,\Gamma; H \;\rightarrow\; \nu''; H'}\;[\textsc{ReplA}]$$

$$\frac{\mathsf{cpy}(H, \nu'') = H', \nu''' \quad H'(\nu) = \kappa^{\nu_{\bot}\,\nu'_{\bot}} \quad \mathsf{index}(\nu', \nu'_{\bot}, H') = \bot \quad \mathsf{reads}(\nu', H') = \texttt{s} \quad H'(\nu_{\bot}) = \texttt{gen}[\overline{\nu}]^{\alpha} \quad H'(\nu'_{\bot}) = \texttt{str}[\overline{\texttt{s}}]^{\alpha'} \quad H'' = H'[\nu_{\bot}/\texttt{gen}[\overline{\nu}\nu''']^{\alpha}][\nu'_{\bot}/\texttt{str}[\overline{\texttt{s}}\texttt{s}]^{\alpha'}]}{\texttt{attr}(\nu, \nu') <\!\!- \nu''\,\Gamma; H \;\rightarrow\; \nu''; H''}\;[\textsc{SetA}]$$

$$\frac{\mathsf{cpy}(H, \nu'') = H', \nu^3 \quad H'(\nu) = \kappa^{\bot\,\bot} \quad \nu^4, \nu^5\,\texttt{fresh} \quad \mathsf{reads}(\nu', H') = \texttt{s} \quad H'' = H'[\nu^4/\texttt{gen}[\nu^3]^{\bot\,\bot}][\nu^5/\texttt{str}[\texttt{s}]^{\bot\,\bot}]}{\texttt{attr}(\nu, \nu') <\!\!- \nu''\,\Gamma; H \;\rightarrow\; \nu''; H''}\;[\textsc{SetB}]$$

Fig. 5. Reduction relation $\rightarrow$

corresponding type with the specified value in it. By default, attributes for these values are empty. A function declaration, [FUN], allocates a closure in the heap and captures the current environment Γ. Variable lookup, [FIND], returns the value of the variable from

the environment. The value of an already evaluated promise is returned by [GETP]. The assignment, [ASS], and super-assignment, [DASS], rules will either define or redefine the target symbol. The value being assigned and all of its attributes are copied recursively. The auxiliary function assign walks the stack and performs the assignment in the first environment that has a binding for the target symbol. If not found, the symbol is added at the top-level. The [GET] rule for array access, $\mathtt{x}[[\nu]]$, is straightforward, it accesses the array at the offset passed as argument. Note that the value returned must be packed in a newly allocated vector of length one of the right type. There are two rules for vector assignment $\mathtt{x}[[\nu]] <- \nu'$. Rule [SETL] applies when the vector is a local variable of the current frame. In that case, the value to be assigned is copied and the assignment is performed in place. Rule [SETG] is more complex. If the variable holding the vector does not occur in the current scope, a new variable will be added to the current scope, the vector is copied with its attributes into the new variable, and finally the assignment is performed.[6] Notice also that all assignment rules yield the right hand side value and not its copy. Finally, there are four rules dealing with attributes. Reading an attribute, $\mathtt{attr}(\nu, \nu')$, uses ν' as a key to find the corresponding value in the attribute vector ([GETA]).[7] The auxiliary function index() returns the index of a string in a vector of strings or $\perp$ if not found. The rules for updating attributes, $\mathtt{attr}(\nu, \nu') <- \nu''$, must consider the two cases. First, when an attribute already exists, the update is done directly ([REPLA]). Second, when an attribute is not present, then the value and name sequences must grow to accommodate the new attribute ([SETA]). Finally, if the attributes are empty, rule [SETB] will create them. It is noteworthy that attributes are modified in place; the objects that they decorate are not copied.

Observations. One of our discoveries while working out the semantics was how eager evaluation of promises turns out to be. The semantics captures this with $\mathbb{C}[]$; the only cases where promises are not evaluated is in the arguments of a function call and when promises occur in a nested function body, all other references to promises are evaluated. In particular, it was surprising and unnecessary to force assignments as this hampers building infinite structures. Many basic functions that are lazy in Haskell, for example, are strict in R, including data type constructors. As for sharing, the semantics cleary demonstrates that R prevents sharing by performing copies at assignments. The R implementation uses copy-on-write to reduce the number of copies. With super-assignment, environments can be used as shared mutable data structures. The way assignment into vectors preserves the pass-by-value semantics is rather unusual and, from personal experience, it is unclear if programmers understand the feature. Extending the semantics to supporting reflection and objects should be possible. Objects are encoded by vectors with attributes that hold their class, as a vector of strings, fields. Methods are functions that abide by a particular naming convention. Dispatch is done by reflecting over defined functions. It is noteworthy that objects are mutable within a function (since fields are attributes), but are copied when passed as an argument.

[6] Our semantics only allows extension of vector at the end. R allows vector to be extended at arbitrary offsets, with missing values added in unused positions.

[7] In R, attributes are represented by a normal vector (values) which, itself, has attributes (names). We simplify the structure for conciseness in the semantics.

5 Corpus Analysis

Given the mix of programming models available to the R user, it is important to understand what features users favor and how they are using those in practice. This section describes the tools we have developed to analyze R programs and the extensive corpus of R programs that we have curated.

5.1 The TraceR Framework

TraceR is a suite of tools for analyzing the performance and characteristics of R code. It consists of three data collection tools built on top of version 2.12.1 of R and several post-processing tools. *TrackeR* generates detailed execution traces, *ProfileR* is a low-overhead profiling tool for the internals of the R VM, and *ParseR* is static analyzer for R code.

TrackeR. To precisely capture user-code behavior, we built TrackeR, a heavily instrumented R VM which records almost every operation executed at runtime. TrackeR's design was informed by our previous work on JavaScript [17]. TrackeR exposes interactions between language features, such as evaluation of promises triggered by function lookups, and how these features are used. It also records promise creation and evaluation, scalar and vector usage, and internally triggered actions (e.g. duplications used for copy-on-write mechanisms). These internal effects are recorded through a mix of trace events and counters. Complex feature interactions such as lazy evaluation and multi-method dispath can result in eager argument evaluation. To capture the triggers for this behavior, prologues are emitted for function calls and associated with the triggering method. Properly tracking the uniqueness of short lived objects, like promises, is complicated by the recycling memory of addresses during garbage collection. R's memory allocations are too large and numerous to use memory maps to resolve this. Instead, a tagging system was used to track the liveness of traced objects. Since, at runtime, function objects are represented as closure with no name, we use R built-in debugging information to map closure addresses to source code. Moreover, control flow can jump between various parts of the call stack when executions are abandoned (e.g. with `tryCatch` or `break` function calls). Keeping the trace consistent requires effort since the implementation of the VM is riddled with calls to `longjmp`. Off-line analysis of traces can quickly exceed machine memory if they are analyzed in-core. Therefore, the tree is processed during its construction and most of it is discarded right away. Specialized trace filters use hooks to register information of interest (e.g. promises currently alive in the system).

ProfileR. While TrackeR reveals program evaluation flow and effects, its heavy instrumentation makes it unsuitable for understanding the runtime costs of language features. For this we built ProfileR, a dedicated counter based profiler which tracks the time costs of operations such as memory management, I/O and foreign calls. Unlike a sampling profiler, ProfileR is precise. It was implemented with care to minimize runtime overheads. The validity of its results was verified against sampling profilers such as oprofile and Apple Instruments. The results are consistent with those tools, and provide more accurate context information. The only notable differences are for very short functions called very frequently, which we avoided instrumenting. R also has a built-in sampling profiler but we found that it did not deliver the accuracy or level of detail we needed.

ParseR. Tracing only yields information on code triggered in a given execution. For a more comprehensive view, ParseR performs static analysis of R programs. It is built on a LL-parser generated with AntLR [14]. Our R grammar seems comprehensive as it parses correctly all R code we could find. Lexical filters can be easily written by using a mixture of tree grammars and visitors. Even though ParseR can easily find accurate grammatical patterns, the high dynamism of R forced us to rely on heuristics when looking for semantic information. ParserR was also used to synchronize the traces generated by TrackeR with actual source code of the programs.

5.2 A Corpus of R Code

We assembled a body of over 3.9 million lines of R code. This corpus is intended to be representative of real-world R usage, but also to help understand the performance impacts of different language features. We classified programs in 5 groups. The *Bioconductor* project open-source repository collects 515 Bioinformatics-related R packages.[8] The *Shootout* benchmarks are simple programs from the Computer Language Benchmark Game[9] implemented in many languages that can be used to get a performance baseline. Some R users donated their code; these programs are grouped under the *Miscellaneous* category. The fourth and largest group of programs was retrieved from the R package archive on CRAN.[10] The last group is the base library that is bundled with the R VM. Fig. 6 gives the size of these datasets. A requirement of all packages in the Bioconductor repository is the inclusion of vignettes. Vignettes are scripts that demonstrate real-world usage of these libraries to potential users. Vignettes also double as simple tests for the programs. They typically come with sample data sets. Out of the 515 Bioconductor programs, we focused on the 100 packages with the longest running vignettes. Some CRAN packages do not have vignettes; this is unfortunate as it makes them harder to analyze. We retained 1 238 out of 3 495 available CRAN packages. It should be noted that while some of the data associated to vignettes are large, they are in general short running.

Name	Bioc.	Shoot.	Misc.	CRAN	Base
# Package	515	11	7	1 238	27
# Vignettes	100	11	4	–	–
R LOC	1.4M	973	1.3K	2.3M	91K
C LOC	2M	0	0	2.9M	50K

Fig. 6. Purdue R Corpus

The Shootout benchmarks were not available in R, so we implemented them to the best of our abilities. They provide tasks that are purely algorithmic, deterministic, and computationally focused. Further, they are designed to easily scale in either memory or computation. For a fair comparison, the Shootout benchmarks stick to the original algorithm. Two out of the 14 Shootout benchmarks were not used because they required multi-threading and one because it relied on highly tuned low-level libraries. We restricted our implementations to standard R features. The only exception is the `knucleotide` problem, where environments served as a substitute for hash maps.

[8] `http://www.bioconductor.org`
[9] `http://shootout.alioth.debian.org/`
[10] `http://cran.r-project.org/`

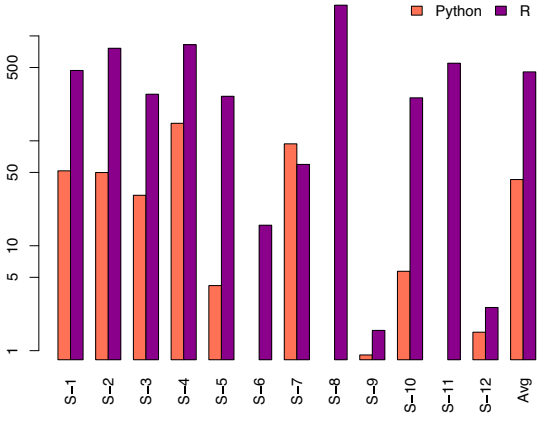
6 Evaluating the R Implementation

Using ProfileR and TraceR, we get an overview of performance bottlenecks in the current implementation in terms of execution time and memory footprint. To give a relative sense of performance, each diagnostic starts with a comparison between R, C and Python using the shootout benchmarks. Beyond this, we used Bioconductor vignettes to understand the memory and time impacts in R's typical usage.

All measurements were made on an 8 core Intel X5460 machine, running at 3.16GHz with the GNU/Linux 2.6.34.8-68 (x86_64) kernel. Version 2.12.1 of R compiled with GCC v4.4.5 was used as a baseline R, and as the base for our tools. The same compiler was used for compiling C programs, and finally Python v2.6.4 was used. During benchmark comparisons and profiling executions, processes were attached to a single core where other processes were denied. Any other machine usage was prohibited.

6.1 Time

We used the Shootout benchmarks to compare the performance of C, Python and R. Results appear in Fig. 7. On those benchmarks, R is on average 501 slower than C and 43 times slower Python. Benchmarks where R performs better, like `regex-dna` (only 1.6 slower than C), are usually cases where R delegates most of its work to C functions.[11]



	Name	Input
S-1	Binary trees	16
S-2	Fankuch redux	10
S-3	Fasta	2.5M
S-4	Fasta redux	2.5M
S-5	K-nucleotide	50K
S-6	Mandelbrot	4K
S-7	N-body	500K
S-8	Pidigits	500
S-9	Regex-dna	2.5K
S-10	Rev. complement	5M
S-11	Spectral norm	640
S-12	Spectral norm alt	11K

Fig. 7. Slowdown of Python and R, normalized to C for the Shootout benchmarks

To understand where time is typically spent, we turn to more representative R programs. Fig. 8 shows the breakdown of execution times in the Bioconductor dataset obtained with ProfileR. Each bar represents a Bioconductor vignette. The key observation is that memory management accounts for an average of 29% of execution time.

[11] For C and Python implementations, we kept the fastest single-threaded implementations. When one was not available, we removed multi-threading from the fastest one. The `pidigits` problem required a rewrite of the C implementation to match the algorithm of the R implementation since the R standard library lacks big integers.

Memory management breaks down into time spent in garbage collection (18%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. Built-in functions are where the true computational work happens, and on average 38% of the execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and one program spends 63% copying arguments. Lookup (4.3% and match 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in Java as they are resolved at compile time. Variable lookup would also be absent in Lisp or Scheme as, once bound, the position of variables in a frame are known. Given the nature of R, many numerical functions are written in C or Fortran; one could thus expect execution time to be dominated by native libraries. The time spent in calls to foreign functions, on average 22%, shows that this is clearly not the case.

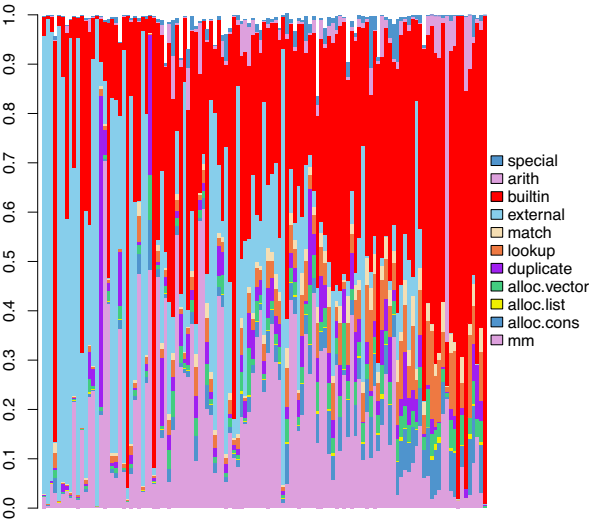


Fig. 8. Breakdown of Bioconductor vignette runtimes as % of total execution time

6.2 Memory

Not only is R slow, but it also consumes significant amounts of memory. Unlike C, where data can be stack allocated, all user data in R must be heap allocated and garbage collected. Fig. 9 compares heap memory usage in C (calls to malloc) and data allocated by the R virtual machine. The R allocation is split between vectors (which are typically user data) and lists (which are mostly used by the interpreter for, e.g., arguments to functions). The graph clearly shows that R allocates orders of magnitude more data than C.

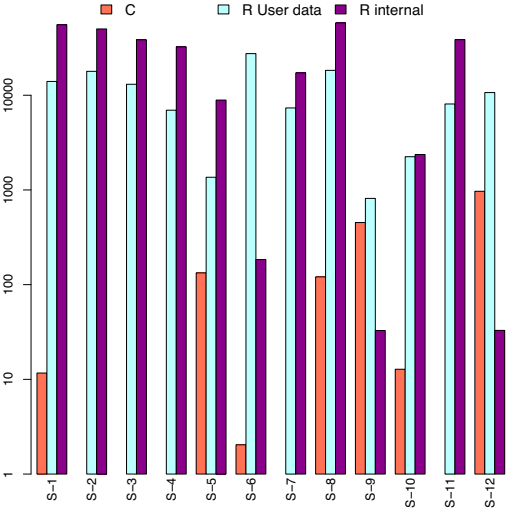


Fig. 9. Heap allocated memory (MB log scale). C vs. R.

In many cases the internal data required is more than the user data. Call-by-value semantics is implemented by a copy-on-write mechanism. Thus, under the covers, function arguments are shared and duplicated when needed. Avoiding duplication reduces memory footprint; on average only 37% of arguments end up being copied. Lists are created by `pairlist` and mostly used by the R VM. In fact, the standard library only has three calls to `pairlist`, the whole CRAN code only eight, and Bioconductor none. The R VM uses them to represent code and to pass and process function call arguments. It is interesting to note that the time spent on allocating lists is greater than the time spent on vectors. Cons cells are large, using 56 bytes on 64-bit architectures, and take up 23 GB on average in the Shootout benchmarks.

Another reason for the large footprint, is that all numeric data has to be boxed into a vector; yet, 36% of vectors allocated by Bioconductor contain only a single number. An empty vector is 40 bytes long. This impacts runtime, since these vectors have to be dereferenced, allocated and garbage collected.

Observations. R is clearly slow and memory inefficient. Much more so than other dynamic languages. This is largely due to the combination of language features (call-by-value, extreme dynamism, lazy evaluation) and the lack of efficient built-in types. We believe that with some effort it should be possible to improve both time and space usage, but this would likely require a full rewrite of the implementation.

7 Evaluating the R Language Design

One of the key claims made repeatedly by R users is that they are more productive with R than with traditional languages. While we have no direct evidence, we will point out that, as shown by Fig. 10, R programs are about 40% smaller than C code. Python is even more compact on those shootout benchmarks, at least in part, because many of the shootout problems are not easily expressed in R. We do not have any statistical

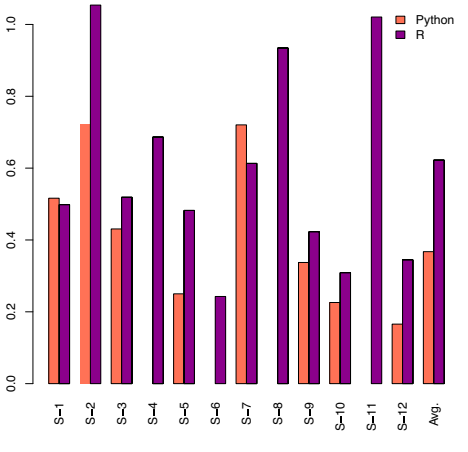


Fig. 10. Shootout Python and R code size, normalized to C

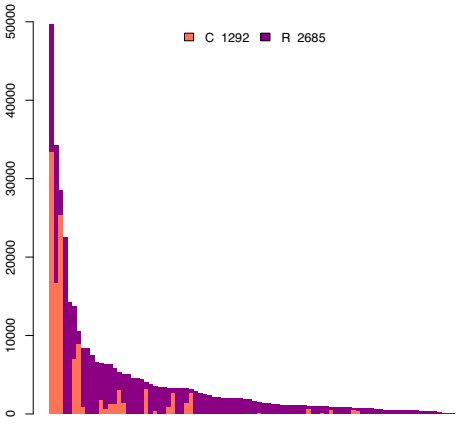


Fig. 11. Bioconductor R and C source code size. (LOC, no comments)

analysis code written in Python and R, so a more meaningful comparison is difficult. Fig. 11 shows the breakdown between code written in R and code in Fortran or C in 100 Bioconductor packages. On average, there is over twice as much R code. This is significant as package developers are surely savvy enough to write native code, and understand the performance penalty of R, yet they would still rather write code in R.

7.1 Functional

Side effects. Assignments can either define or update variables. In Bioconductor, 45% of them are definitions, and only two out of 217 million assignments are definitions in a parent frame by super assignment. In spite of the availability of non-local side effects (i.e., `<<-`), 99.9% of side effects are local. Assignments done through functions such as `[]<-` need an existing data structure to operate on, thus they are always side effecting. Overall they account for 22% of all side effects and 12% of all assignments.

Scoping. R symbol lookup is context sensitive. This feature, which is neither Lisp nor Scheme scoping, is exercised in less than 0.05% of function name lookups. However, even though this number is low, the number of symbols actually checked is 3.6 on average. The only symbols for which this feature actually mattered in the Bioconductor vignettes are `c` and `file`, both popular variables names and built-in functions.

Parameters. The R function declaration syntax is expressive and this expressivity is widely used. In 99% of the calls, at most 3 arguments are passed, while the percentage of calls with up to 7 arguments is 99.74% (see Fig. 12). Functions that are close to this average are typically called with positional arguments. As the number of parameters increases, users are more likely to specify function parameters by name. Similarly, variadic parameters tend to be called with large numbers of arguments.

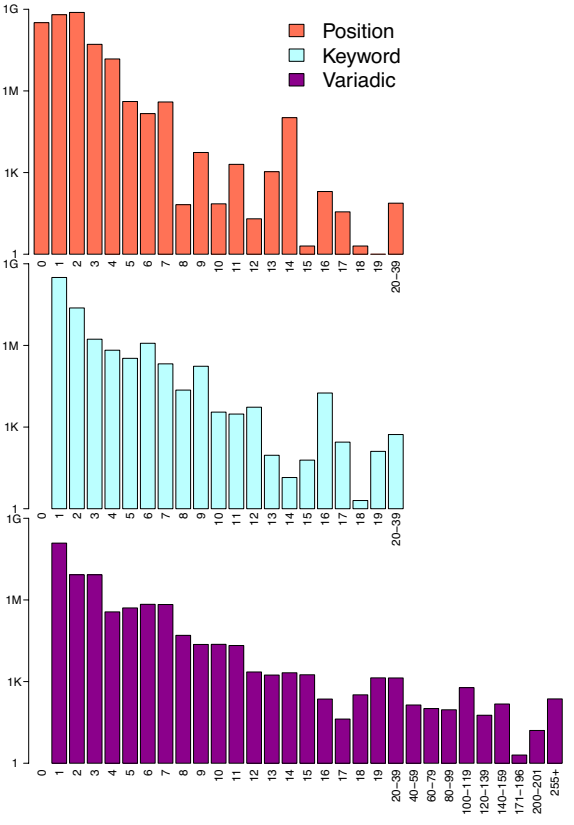


Fig. 12. Histogram of the number of function arguments in Bioconductor. (Log scale)

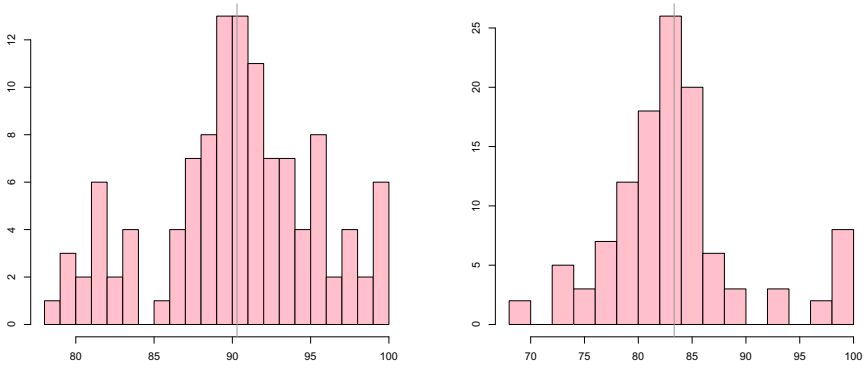
	Bioc		Shootout		Misc		CRAN	Base
	stat.	dyn.	stat.	dyn.	stat.	dyn.	stat.	stat.
Calls	1M	3.3M	657	2.6G	1.5K	10.0G	1.7G	71K
by keyword	197K	72M	67	10M	441	260M	294K	10K
# keywords	406K	93M	81	15M	910	274M	667K	18K
by position	1.0M	385M	628	143M	1K	935M	1.6G	67K
# positional	2.3M	6.5G	1K	5.2G	3K	18.7G	3.5G	125K

Fig. 13. Number of calls by category

Fig. 13 gives the number of calls in our corpus and the total number of keyword and variadic arguments. Positional arguments are most common between 1 and 4 arguments, but are used all the way up to 25 arguments. Function calls with between 1 and 22 named arguments have been observed. Variadic parameters are used to pass from 1 to more than 255 arguments. Given the performance costs of parsing parameter lists in the current implementation, it appears that optimizing calling conventions for function of four parameters or less would greatly improve performance. Another interesting consequence of the prevalent use of named parameters is that they become part of the interface of the function, so alpha conversion of parameter names may affect the behavior of the program.

Laziness. Lazy evaluation is a distinctive feature of R that has the potential for reducing unnecessary work performed by a computation. Our corpus, however, does not bear this out. Fig. 14(a) shows the rate of promise evaluation across all of our data sets. The average rate is 90%. Fig. 14(b) shows that on average 80% of promises are evaluated in the first function they are passed into. In computationally intensive benchmarks the rate of promise evaluation easily reaches 99%. In our own coding, whenever we encountered higher rates of unevaluated promises, finding where this occurred and refactoring the code to avoid those promises led to performance improvements.

Promises have a cost even when not evaluated. Their cost in in memory is the same as a pairlist cell, i.e., 56 bytes on a 64-bit architecture. On average, a program allocates 18GB for them, thus increasing pressure on the garbage collector. The time cost of promises is roughly one allocation, a handful of writes to memory. Moreover, it is a data



(a) Promises evaluated (in %) (b) Promises evaluated on same level (in %)

Fig. 14. Promises evaluation in all data sets. The y-axis is the number of programs.

type which has to be dispatched and tested to know if the content was already evaluated. Finally, this extra indirection increases the chance of cache misses. An example of how unevaluated promises arise in R code is the `assign` function, which is heavily used in Bioconductor with 23 million calls and 46 million unevaluated promises.

```
function(x,val,pos=-1,env=as.environment(pos),immediate=TRUE)
    .Internal(assign(x,val,env))
```

This function definition is interesting because of its use of dependent parameters. The body of the function never refers to `pos`, it is only used to modify the default value of `env` if that parameter is not specified in the call. Less than 0.2% of calls to `assign` evaluate the promise associated with `pos`.

It is reasonable to ask if it would be valid to simply evaluate all promises eagerly. The answer is unfortunately no. Promises that are passed into a function which provides a language extension may need special processing. For instance in a loop, promises are intended to be evaluated multiple times in an environment set up with the right variables. Evaluating those eagerly will result in a wrong behavior. However, we have not seen any evidence of promises being used to extend the language outside of the base libraries. We infer this from calls to the `substitute` and `assimilate` functions. Another possible reason for not switching the evaluation strategy is that promises perform and observe side effects.

```
x <- y <- 0
fun <- function(a, b) if(runif(1)>.5) a+b else b+a
fun(x<-y+1, y<-x+2) # Result is always a+b, but can be either 4 or 5
```

This code snippet will yield different results depending on the order the two promises passed to `fun` are going to be evaluated. Taking into account the various oddities of R, such as lookups that force evaluation of all promises in scope, it is reasonable to wonder if relying on a particular evaluation order is a wise choice for programmers.

7.2 Dynamic

Eval. The `eval` function is widely used in R code with 8 500 static calls in CRAN and 5 800 calls in Bioconductor. The total number of dynamic calls in our benchmarks was 2.2 million. These are rather large numbers. We focus on the 15 call sites where each represents more than 1% of the total dynamic calls. Together these call sites account for 88% of `eval`. The `match.arg` function is the highest user with 54% of all calls to `eval`. In the 14 other call sites to `eval`, we see two uses cases. The most common is the evaluation of the source code of a promise retrieved by `substitute` in a new environment. This is done in the `with` function. The other use case is the invocation of a function whose name or arguments are determined dynamically. For this purpose, R provides `do.call`, and thus using `eval` is overkill.

Substitute. Promises provide a kind of limited automatic quoting of arguments as the `substitute` function can retrieve the textual representation of the source expression of any promise. A typical use case is to add a legend to a chart when no text is provided; this is done by retrieving the expression passed to the plot function and using it as a

legend. However, this usage is limited to one level of nesting, and passing a promise to another function will destroy that information.

```
f <- function(x) substitute(x)
b <- function(x) (function(y) substitute(y))(x)
f(1 + 1) # 1 + 1
b(1 + 1) # x
```

The example above shows that `substitute` only retrieves the text of the last argument. In Bioconductor, `substitute` is called from 51 call sites 3.6 million times, but only 11 call sites are in user code (the rest comes from the standard library). They account for 2% of dynamic calls.

match.arg. The `match.arg` function takes arguments `arg` and `choices`; it matches `arg` against a table of candidate values as specified by `choices`. In practice, 75% of the calls to this function only pass the first argument, like the following code snippet:

```
magic <- function(type=c("mean","median","trimmed"))
            return (match.arg(type))
```

A call to `magic("t")` returns `trimmed`. Notice that the string `trimmed` only occurs in the default argument which is not used in this case as a value of `"t"` is provided. What happens is that `match.arg` reaches into its caller, reflectively finds the default value for `type` and uses it as the value of `choices`; this is done as follows,

```
match.arg <- function (arg, choices)
  if (missing(choices)) {
    formal.args <- formals(sys.function(sys.parent()))
    choices <- eval(formal.args[[deparse(substitute(arg))]])
  ...
```

The first line checks if the `choices` argument was passed to the function. The second line gathers the list of parameters of the caller. Finally, the last line extracts from this list the parameter that has the same name as `arg` and `eval`s it.

Environments. Explicit environment manipulation hinders compiler optimizations. In our benchmarks these functions are called often. But it turns out that they are most often used to short-circuit the by-value semantics of R. We discovered that 87% of the calls to `remove`, which deletes a local variable from the current frame, are used as part of an implementation of a hash map. R also allows programs to change the nesting of an environment with `parent.env`. But 69% of these changes are used by the R VM's lazy load mechanism, and 30% by the `proto` library which implements prototypes [20] and uses environments to avoid copies.

7.3 Objects

The S4 object model has been promoted as a replacement for S3 by parts of the R community [16,3]. However, our numbers do show this happening. Thirteen years after the introduction of S4, S3 classes still dominate. From the number of methods introduced and the number of times they are redefined, S3 classes seem to be used quite differently than S4 classes.

Fig. 15 summarizes the use of object-orientation in the corpus. In our corpus, 1 055 S3 classes, or roughly one fourth of all classes, have no methods defined on them and 1 107 classes, 30%, have only a `print` or `plot` method. Fig. 16 gives the number of redefinitions of S3 methods. Any number of definitions larger than one suggest some polymorphism. Unsurprisingly, `plot` and `print` dominate. While important, does the need for these two functions really justify an object system? Attributes already allow the programmer to tag values, and could easily be used to store closures for a handful of methods like `print` and `plot`. A prototype-based system would be simpler and probably more efficient than the S3 object system. Finally, only 30% of S3 classes are really object-oriented. This translates to one class for every two packages. This is quite low and makes rewriting them as S4 objects seem feasible. Doing so could simplify and improve both R code and the evaluator code.

		Bioc	Misc	CRAN	Base	Total
S3	# classes	1 535	0	3 351	191	3 860
	# methods	1 008	0	1 924	289	2 438
	Avg. redef.	6.23	0	7.26	4.25	9.75
	Method calls	13M	58M	-	-	76M
	Super calls	697K	1.2M	-	-	2M
S4	# classes	1 915	2	1 406	63	2 893
	# singleton	608	2	370	28	884
	# leaves	819	0	621	16	1 234
	Hier. depth	9	1	8	4	9
	Direct supers	1.09	0	1.13	0.83	1.07
	# methods	4 136	22	2 151	24	5 557
	Avg. redef.	3	1	3.9	2.96	3.26
	Redef. depth	1.12	1	1.21	1.08	1.14
	# new	668K	64	-	-	668K
	Method calls	15M	266	-	-	15M
	Super calls	94K	0	-	-	94K

Fig. 15. Object usage in the corpus

S4 objects on the other hand, seem to be used in a more traditional way. The class hierarchies are not deep (maximum is 9), however they are not flat either. The number of parent classes is surprisingly low (see [5] for comparison), but reaches a maximum of 50 direct superclasses. In Fig. 15, singleton classes, i.e., classes which are both root and leaf, are ignored. At first glance, the number of method redefinitions seems to be a bit smaller than what we find in other object languages. This is partially explained by the absence of a root class, the use of class unions, and because multi-methods are declared outside of classes. The number of redefinitions, i.e., one method applied to a more specific class, is very low (only 1 in 25 classes). This pattern suggests that the S4 object model is mostly used to overcome an absence of structure declarations rather than to add objects in statistical computing. Even when biased by Bioconductor, which pushes for S4 adoption, the use

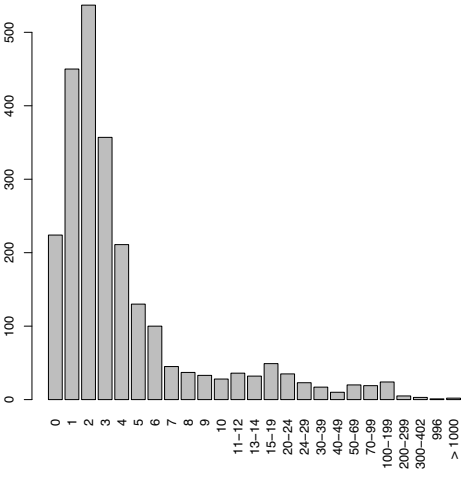


Fig. 16. S3 method redefinitions (on x axis)

of S4 classes remains low. Part of the reason may be the perception that S3 classes are less verbose and clumsy to write than S4; it may also come from the fact that the base libraries use S3 classes intensively and this is reflected in our data.

7.4 Experience

Implementing the shootout problems highlighted some limitations of R. The pass-by-value semantics of function calls is often cumbersome. The R standard library does not provide data structures such as growable arrays or hash maps found in many other languages. Implementing them efficiently is difficult without references. To avoid copying, we were constrained to either use environments as a workaround, or to inline these operations and then make use of scoping and `<<-` as needed. Either choice makes the code unnecessarily verbose and readability suffers. Moreover, the former choice brings the question of whether the environments are used as intended, and the latter has a serious impact on code maintenance.

We found performance hard to predict. Without a solid understanding of the implementation, users are bound to be surprised by the impact of seemingly small changes to their code. In the binary tree program, adding an extra (and unneeded) `return` statement or a pair of parentheses `()` will impact performance in a noticeable way. Fig. 17 shows impact of adding these operations on performance for different input sizes.

Input size	12	13	14
Base time	6s	12s	31s
`()`	+7.4%	+5.3%	+6.3%
`return`	+4.6%	+5.4%	+5.4%

Fig. 17. Adding overheads

8 Conclusions

This paper reports on our investigations into the design and implementation of the R language. Despite having millions of users and being, in many respects, a success story, R has received little attention from our community. With the exception of [13], which mistakenly characterized R as strict and imperative, ours is the first attempt to introduce R to a mainstream computer science audience.

Our first challenge was to understand the unconventional semantics of the language and the sometimes subtle interactions between its features. While some documentation exists, it is incomplete. The language is effectively defined by the successive releases of its implementation. Relying on an implementation as the authoritative specification of a language is unsatisfactory; the R interpreter is constrained by implementation decisions and presents a programming model that is at same time overconstrained and ambiguous. Implementation details are exposed and slowly bleed into the language. We have found it useful, for our own sake, to formalize the current implementation of R, focusing on features such as lazy evaluation, variable scoping and binding, and copy-semantics. Even though our semantics does not cover all of R, we did not oversimplify. We present a proper subset of R; we are confident that we will be able to extend it to larger portions of the language. As a language, R is like French; it has an elegant core, but every rule comes with a set of ad-hoc exceptions that directly contradict it.

A language definition is only part of the picture. The next question is how it is used in practice. Even the most elegant feature can be misused, and the ugliest language design

can be used well when sufficient discipline is employed. To understand R in the wild, we implemented the TraceR framework and gathered over 3 million lines of code from various sources to form the largest open source R benchmark suite. Armed with these tools we started looking at how the exotic features of R are used by programs and what are the overheads and costs involved in supporting those features.

The R user community roughly breaks down into three groups. The largest groups are the end users. For them, R is mostly used interactively and R scripts tend to be short sequences of calls to prepackaged statistical and graphical routines. This group is mostly unaware of the semantics of R, they will, for instance, not know that arguments are passed by copy or that there is an object system (or two). The second, smaller and more savvy, group is made up of statisticians who have a reasonable grasp of the semantics but, for instance, will be reluctant to try S4 objects because they are "complex". This group is responsible for the majority of R library development. The third, and smallest, group contains the R core developers who understand both R and the internals of the implementation and are thus comfortable straddling the native code boundary.

One of the reasons for the success of R is that it caters to the needs of the first group, end users. Many of its features are geared towards speeding up interactive data analysis. The syntax is intended to be concise. Default arguments and partial keyword matches reduce coding effort. The lack of typing lowers the barrier to entry, as users can start working without understanding any of the rules of the language. The calling convention reduces the number of side effects and gives R a functional flavor. But, it is also clear that these very features hamper the development of larger code bases. For robust code, one would like to have less ambiguity and would probably be willing to pay for that by more verbose specifications, perhaps going as far as full-fledged type declarations. So, R is not the ideal language for developing robust packages. Improving R will require increasing encapsulation, providing more static guarantees, while decreasing the number and reach of reflective features. Furthermore, the language specification must be divorced from its implementation and implementation-specific features must be deprecated.

The balance between imperative and functional features is fascinating. We agree with the designers of R that a purely functional language whose main job is to manipulate massive numeric arrays is unlikely to be a success. It is simply too useful to be able to perform updates and have a guarantee that they are done in place rather than hope that a smart compiler will be able to optimize them. The current design is a compromise between the functional and the imperative; it allows local side effects, but enforces purity across function boundaries. It is unfortunate that this simple semantics is obscured by exceptions such as the super-assignment operator (`<<-`) which is used as a sneaky way to implement non-local side effects.

One of the most glaring shortcomings of R is its lack of concurrency support. Instead, there are only native libraries that provide behind-the-scenes parallel execution. Concurrency is not exposed to R programmers and always requires switching to native code. Adding concurrency would be best done after removing non-local side effects, and requires inventing a suitable concurrent programming model. One intriguing idea would be to push on lazy evaluation, which, as it stands, is too weak to be of much use outside of the base libraries, but could be strengthened to support parallel execution.

The object-oriented side of the language feels like an afterthought. The combination of mutable objects without references or cyclic structures is odd and cumbersome. The simplest object system provided by R is mostly used to provide printing methods for different data types. The more powerful object system is struggling to gain acceptance.

The current implementation of R is massively inefficient. We believe that this can, in part, be ascribed to the combination of dynamism, lazy evaluation, and copy semantics, but it also points to major deficiencies in the implementation. Many features come at a cost even if unused. That is the case with promises and most of reflection. Promises could be replaced with special parameter declarations, making lazy evaluation the exception rather than the rule. Reflective features could be restricted to passive introspection which would allow for the dynamism needed for most uses. For the object system, it should be built-in rather than synthesized out of reflective calls. Copy semantics can be really costly and force users to use tricks to get around the copies. A limited form of references would be more efficient and lead to better code. This would allow structures like hash maps or trees to be implemented. Finally, since lazy evaluation is only used for language extensions, macro functions à la Lisp, which do not create a context and expand inline, would allow the removal of promises.

Acknowledgments. The authors benefited from encouragements, feedback and comments from John Chambers, Michael Haupt, Ryan Macnak, Justin Talbot, Luke Tierney, Gaël Thomas, Olga Vitek, Mario Wolczko, and the reviewers. This work was supported by NSF grant OCI-1047962.

References

1. Becker, R.A., Chambers, J.M., Wilks, A.R.: The New S Language. Chapman and Hall (1988)
2. Bobrow, D.G., Kahn, K.M., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F.: In: Conference on Object-Oriented Programming, Languages and Applications, OOPSLA (1986)
3. Chambers, J.M.: Software for Data Analysis: Programming with R. Springer (2008)
4. Chambers, J.M., Hastie, T.J.: Statistical Models in S. Chapman & Hall (1992)
5. Ducournau, R.: Coloring, a Versatile Technique for Implementing Object-Oriented Languages. Software: Practice and Experience 41(6), 627–659 (2011)
6. Kent Dybvig, R.: The Scheme Programming Language. MIT Press (2009)
7. Gentleman, R., et al. (eds.): Bioinformatics and Computational Biology Solutions Using R and Bioconductor. Statistics for Biology and Health. Springer (2005)
8. Gentleman, R., Ihaka, R.: Lexical scope and statistical computing. Journal of Computational and Graphical Statistics 9, 491–508 (2000)
9. Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of Haskell: being lazy with class. In: Conference on History of programming languages, HOPL (2007)
10. Ihaka, R., Gentleman, R.: R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics 5(3), 299–314 (1996)
11. Keele, L.: Semiparametric Regression for the Social Sciences. Wiley (2008)
12. Kiczales, G., Rivieres, J.D., Bobrow, D.G.: The Art of the Metabobject Protocol: The Art of the Metaobject Protocol. MIT Press (1991)
13. Mitchell, E.G.: Functional programming through deep time: modeling the first complex ecosystems on earth. In: Conference on Functional Programming, ICFP (2011)
14. Parr, T., Fisher, K.: Ll(*): the foundation of the Antlr parser generator. In: Conference on Programming Language Design and Implementation, PLDI (2011)

15. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing (2011)
16. R Development Core Team: The R language definition. R Foundation for Statistical Computing, http://cran.r-project.org/doc/manuals/R-lang.html
17. Richards, G., Lesbrene, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: Conference on Programming Language Design and Implementation, PLDI (2010)
18. Smith, D.: The R ecosystem. In: The R User Conference 2011 (August 2011)
19. Steele Jr., G.L.: Common LISP: the language, 2nd edn. Digital Press (1990)
20. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Conference on Object-Oriented Programming, Languages and Applications, OOPSLA (1987)
21. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115, 38–94 (1992)

$$\frac{H(\nu) = \mathtt{num}[\mathtt{n}]^{\alpha}}{\mathsf{reads}(\nu, H) = \mathtt{n}}\ [\textsc{reads}] \qquad \frac{H(\nu) = \mathtt{str}[\mathtt{s}]^{\alpha}}{\mathsf{readn}(\nu, H) = \mathtt{s}}\ [\textsc{readb}]$$

$$\frac{H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad \nu'\ \mathtt{fresh} \quad H' = H[\nu'/\mathtt{num}[\mathtt{n}_{\mathtt{m}}]^{\perp\,\perp}]}{\mathsf{get}(\nu, \mathtt{m}, H) = \nu', H'}\ [\textsc{GetN}]$$

$$\frac{H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad \nu'\ \mathtt{fresh} \quad H' = H[\nu'/\mathtt{str}[\mathtt{s}_{\mathtt{m}}]^{\perp\,\perp}]}{\mathsf{get}(\nu, \mathtt{m}, H) = \nu', H'}\ [\textsc{GetS}] \qquad \frac{H(\nu) = \mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}} \ldots]^{\alpha}}{\mathsf{get}(\nu, \mathtt{m}, H) = \nu_{\mathtt{m}}, H'}\ [\textsc{GetG}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n} \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetN}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s} \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetS}]$$

$$\frac{H(\nu) = \mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[\nu_1 \ldots \nu' \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetG}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}\,\mathtt{n}]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetNe}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}\,\mathtt{s}]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetSe}]$$

$$\frac{H(\nu) = \mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}}\,\nu']^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetGe}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[s(\mathtt{n}_1) \ldots \mathtt{s} \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetNs}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[g(\mathtt{n}_1) \ldots \nu' \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetNg}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots s(\mathtt{n}) \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetSn}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[g(\mathtt{s}_1) \ldots \nu' \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetSg}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[s(\mathtt{n}_1) \ldots s(\mathtt{n}_{\mathtt{m}})\,\mathtt{s}]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetNse}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[g(\mathtt{n}_1) \ldots g(\mathtt{n}_{\mathtt{m}})\,\nu']^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetNge}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}\,s(\mathtt{n})]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetSne}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[g(\mathtt{s}_1) \ldots g(\mathtt{s}_{\mathtt{m}})\,\nu']^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetSge}]$$

$$\frac{H(\iota) = F \quad F(\mathtt{x}) = \nu}{\iota(H, \mathtt{x}) = \nu}\ [\textsc{Look0}] \qquad \frac{\Gamma = \iota * \Gamma' \quad \iota(H, \mathtt{x}) = \nu}{\Gamma(H, \mathtt{x}) = \nu}\ [\textsc{Look1}] \qquad \frac{\Gamma = \iota * \Gamma' \quad H(\iota) = F \quad \mathtt{x} \notin dom(F) \quad \Gamma'(H, \mathtt{x}) = \nu}{\Gamma(H, \mathtt{x}) = \nu}\ [\textsc{Look2}]$$

$$\frac{\mathsf{cpy}(H, \nu_\perp) = H', \nu''_\perp \quad \mathsf{cpy}(H', \nu'_\perp) = H'', \nu'''_\perp}{\mathsf{cpy}(H, \nu_\perp, \nu'_\perp) = H'', \nu''_\perp, \nu'''_\perp}\ [\textsc{Copy0}] \qquad \frac{}{\mathsf{cpy}(H, \perp) = H, \perp}\ [\textsc{Copy1}]$$

$$\frac{H(\nu) = \kappa^{\alpha} \quad \alpha = \nu_\perp \nu'_\perp \quad \mathsf{cpy}(H, \nu_\perp, \nu'_\perp) = H'\,\nu''_\perp, \nu'''_\perp \quad \nu''\ \mathtt{fresh} \quad H'' = H'[\nu''/\kappa^{\nu''_\perp\,\nu'''_\perp}]}{\mathsf{cpy}(H, \nu) = H'', \nu''}\ [\textsc{Copy2}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad H(\iota) = F \quad \mathtt{x} \in dom(F) \quad F' = F[\mathtt{x}/\nu] \quad H' = H[\iota/F']}{\mathsf{assign}(\mathtt{x}, \nu, \Gamma, H) = H'}\ [\textsc{Super1}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad H(\iota) = F \quad \mathtt{x} \notin dom(F) \quad \mathsf{assign}(\mathtt{x}, \nu, \Gamma', H) = H'}{\mathsf{assign}(\mathtt{x}, \nu, \Gamma, H) = H'}\ [\textsc{Super2}]$$

$$\frac{\Gamma = \iota \quad H(\iota) = F \quad F' = F[\mathtt{x}/\nu] \quad H' = H[\iota/F']}{\mathsf{assign}(\mathtt{x}, \nu, \Gamma, H) = H'}\ [\textsc{Super3}]$$

Fig. 18. Auxiliary definitions

McSAF: A Static Analysis Framework for MATLAB*

Jesse Doherty and Laurie Hendren

School of Computer Science, McGill University, Montreal, QC, Canada
{jesse,hendren}@cs.mcgill.ca

Abstract. MATLAB is an extremely popular programming language used by scientists, engineers, researchers and students world-wide. Despite its popularity, it has received very little attention from compiler researchers. This paper introduces McSaf, an open-source static analysis framework which is intended to enable more compiler research for MATLAB and extensions of MATLAB. The framework is based on an intermediate representation (IR) called McLast, which has been designed to capture all the key features of MATLAB, while at the same time being simple for program analysis. The paper describes both the IR and the procedure for creating the IR from the higher-level AST. The analysis framework itself provides visitor-based traversals including fixed-point-based traversals to support both forwards and backwards analyses. McSaf has been implemented as part of the McLab project, and the framework has already been used for a variety of analyses, both for MATLAB and the AspectMatlab extension.

1 Introduction

MATLAB is a popular dynamic ("scripting") programming language that has been in use since the late 1970s, and a commercial product of MathWorks since 1984, with millions of users in the scientific, engineering and research communities.[1] There are currently over 1200 books based on MATLAB and its companion software, Simulink (`http://www.mathworks.com/support/books`).

Despite the popularity of the language, there exists relatively little compiler research for MATLAB, and without an existing framework it is difficult for researchers to tackle such research. McSaf, the topic of this paper, is a compiler analysis framework that is intended to enable compiler research by providing both a convenient intermediate representation and an intraprocedural analysis framework which can be used both for MATLAB and language extensions of MATLAB. It has been developed as a key component of the McLab project [3].

* This work was supported, in part, by NSERC. A special thanks to Soroush Radpour for his help with the experiments.

[1] The most recent data from MathWorks shows that the number of users of MATLAB was 1 million in 2004, with the number of users doubling every 1.5 to 2 years.(From `www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf`.)

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 132–155, 2012.

This paper does not just report on a standard compiler engineering effort. Designing an intermediate representation (IR) for MATLAB that is suitable for program analysis was quite challenging for three reasons. First, the MATLAB language has grown somewhat organically and does not have a precise documented semantics. Thus, the IR needs to expose those semantics both correctly and in a manner that simplifies subsequent analyses. One example is the correct handling of the "`&`" and "`|`" operators, which are short-circuiting in some situations and not in others. Secondly, MATLAB supports several high-level features that are convenient for programmers, but difficult for compilers. For example, an assignment statement may assign to numerous variables at the same time, and the number of such variables may not be known statically. The IR must expose and simplify such features. Thirdly, the IR must correctly encode which identifiers refer to variables, and which refer to named functions. This is not trivial since the syntax of MATLAB is ambiguous, and there are no explicit declarations of variables. For example, the expression "`a(i)`" could mean four different things, depending on whether "`a`" is a variable or function, and whether "`i`" is a variable or a function. Consider two of the cases. If both "`a`" and "`i`" are variables, the expression "`a(i)`" means the i^{th} element of array "`a`". If both "`a`" and "`i`" are named functions, then "`a(i)`" is a call to function "`a`", where the argument is a call to the built-in function "`i`" (which returns the complex value i). Clearly a program analysis will apply very different analysis rules for an array access or a function call. Thus, these syntactic ambiguities should be resolved in the IR.

Given a suitable IR, another important challenge is to design an analysis framework that supports a wide variety of intraprocedural analyses. Our goal was to design a framework that is easy to use, thus enabling others to design new analyses. We also felt that it was important that our approach should support language extensions so that an analysis originally designed for MATLAB could be easily adapted to an extension of MATLAB.

In this paper we provide our solution to these challenges. Our contributions are as follows:

Design of McLAST IR: The design and implementation of McLAST, a lower-level AST-based intermediate representation that both exposes important MATLAB semantics and simplifies program analysis.

Generating McLAST: The design and implementation of a collection of simplifying transformations that together provide a mechanism for generating the McLAST IR from the higher-level AST. This simplification framework encodes the dependency structure between simplifications to enable the application of a subset of simplifications and also to allow for a structured way of introducing future simplifications. The simplifying transformations themselves support both standard and MATLAB-specific transformations.

Analysis Framework: The design and implementation of an extensible program analysis framework for MATLAB. The framework supports a variety of visitor-based traversal mechanisms including a depth-first style traversal suitable for flow-insensitive analyses and a structured-program-analysis

traversal which supports both forward and backwards flow-sensitive analyses. The flow-sensitive traversal mechanism automatically supports conditional control flow, and iterative constructs, including support for "`break`" and "`continue`" statements.

Extensibility: Both the simplification and analysis frameworks have been designed to support extensions of MATLAB.

The remainder of this paper is structured as follows. In Section 2 we provide an overview of the whole MCLAB project and show how the MCSAF framework fits into the big picture. Section 3 introduces the MCLAST IR and the simplifying transformations, while Section 4 describes the analysis framework. Related work is discussed in Section 5 and Section 6 concludes.

2 Overview

The MCSAF system forms a key component of the MCLAB project as illustrated in Figure 1. As a whole, MCLAB is intended as a complete toolkit for MATLAB systems including an extensible front-end built using MetaLexer [5] and JastAdd [2,12], and several back-ends including source-to-source tools like the refactoring toolkit [18], static compilers such as the MATLAB-to-FORTRAN translator (MCFOR) [16] and the MATLAB Tamer [10], and dynamic virtual machine/JIT systems (MCVM) [6]. All parts of MCLAB, with the exception of MCVM, are implemented using Java and Java-based tools.

MCSAF is the heart of the system, providing both the lower-level MCLAST IR and the analysis toolkit/engine (as indicated by the grey boxes). The dotted boxes in Figure 1 denote existing implementations of flow analyses that use the MCSAF toolkit. Note that MCSAF can be used to build analyses both on the higher-level AST (for example, kind analysis), as well as on the lower-level MCLAST. MCSAF includes three different implementations of kind analysis [9], and a variety of standard dataflow analyses. Other components of the larger MCLAB system also build analyses using MCSAF. The refactoring toolkit implements a variety of specialized analyses to enable refactoring transformations [18], and the MATLAB Taming toolkit implements specialized simplifications and analyses to generate a call graph and type/class information [10]. MCVM uses both the IR and some standard flow analysis information generated by MCSAF.[2]

It is our intention that future users of MCSAF would add to both the standard flow analyses and develop new specialized analyses for both our existing backends and for their own projects/tools.

3 Intermediate Representations and Simplifications

One of the key steps in designing MCSAF was to design and implement an appropriate intermediate representation which exposed key semantics of MATLAB

[2] Specialized analyses in McVM are implemented in C++ and LLVM [15], and thus do not use MCSAF.

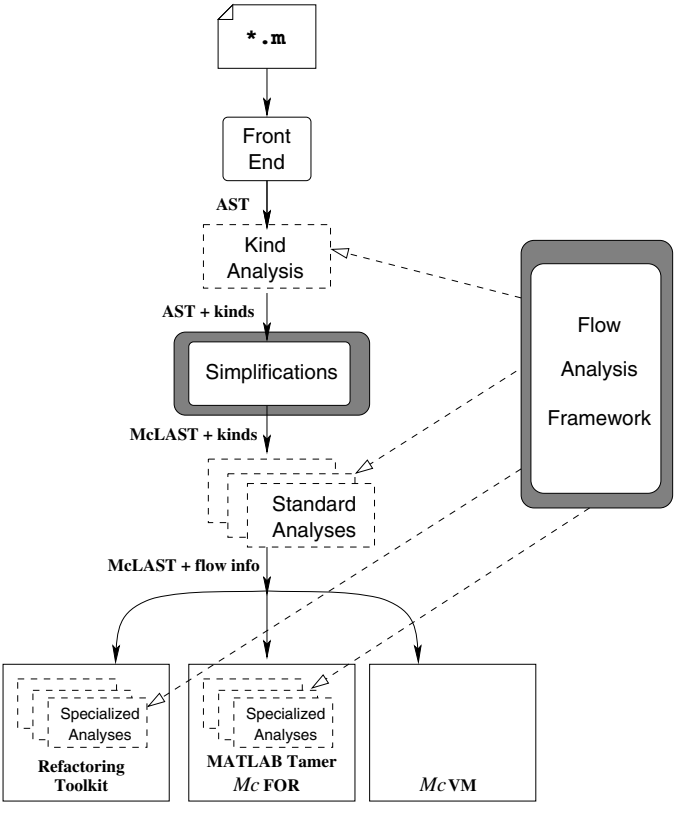


Fig. 1. Overview of McLab and McSaf. The grey boxes indicate the components presented in this paper: *Simplifications* are discussed in Section 3 and the *Flow Analysis Framework* is presented in Section 4. The dotted boxes correspond to existing analyses implemented using McSaf.

and provided a good basis for analyses. Although we hope the end result is quite clean and accomplishes these goal, the process of determining the correct IR was not at all straightforward. In this section we introduce the existing high-level AST and then discuss the simplifications we developed that result in the lower-level IR, McLast.

3.1 High-Level AST

The McLab front-end already has a well-defined AST specification and a Java implementation of the AST that is generated based on a JastAdd specification. Figure 2 gives an extract of the AST specification.

For those not familiar with JastAdd, we give a quick introduction. Each declaration defines a node type, which may be *abstract* or *concrete*. An abstract

```
// Top−level structure
abstract Program;
CompilationUnits ::= Program*;
Script : Program ::= HelpComment* Stmt*;
FunctionList : Program ::= Function*;
Function ::= OutputParam:Name* <Name:String> InputParam:Name*
    HelpComment* Stmt* NestedFunction:Function*;

// Statements
abstract Stmt;
WhileStmt : Stmt ::= Expr Stmt*;
ForStmt : Stmt ::= AssignStmt Stmt*;
BreakStmt : Stmt;
ContinueStmt : Stmt;
ReturnStmt : Stmt;
IfStmt : Stmt ::= IfBlock* [ElseBlock];
IfBlock ::= Condition:Expr Stmt*;
ElseBlock ::= Stmt*;
SwitchStmt : Stmt ::= Expr SwitchCaseBlock* [DefaultCaseBlock];
SwitchCaseBlock ::= Expr Stmt*;
DefaultCaseBlock ::= Stmt*;
AssignStmt : Stmt ::= LHS:Expr RHS:Expr;
ExprStmt : Stmt ::= Expr;

// Expressions
abstract Expr;
abstract LiteralExpr : Expr;
abstract LValueExpr : Expr;
abstract UnaryExpr : Expr ::= Operand:Expr;
abstract BinaryExpr : Expr ::= LHS:Expr RHS:Expr;
NameExpr : LValueExpr ::= Name;
ParameterizedExpr : LValueExpr ::= Target:Expr Arg:Expr*;
CellIndexExpr : LValueExpr ::= Target:Expr Arg:Expr*;
DotExpr : LValueExpr ::= Target:Expr Field:Name;
MatrixExpr : LValueExpr ::= Row*;
Name ::= <ID : String>;
Row ::= Element:Expr*;
 ...
RangeExpr : Expr ::= Lower:Expr [Incr:Expr] Upper:Expr;
ColonExpr : Expr;
EndExpr : Expr;
CellArrayExpr : Expr ::= Row*;
FunctionHandleExpr : Expr ::= Name;
LambdaExpr : Expr ::= InputParam:Name* Body:Expr;
```

Fig. 2. AST Definition

declaration, such as in line 2, will result in an abstract Java class being generated. Declarations can declare subtypes, for example on lines 4 and 5, Script and FunctionList are declared to be subtypes of Program. Each declaration may list children, for example, lines 6-7 declare that a Function has six children. Children may be explicitly named or not. For example, the first child of Function has the name OutputParam, whereas the fourth and fifth children are not named. A child may be a list of a specific type (indicated by $*$), a singleton, or optional (indicated by [...]).

The AST definition follows the natural abstract syntax of MATLAB. A MATLAB program consists of a collection of compilation units. Each compilation unit can contain either a script or a list of functions. Scripts contain only comments and statements. Functions have output parameters, input parameters, and possibly nested functions. The function body consists of comments and statements. Statements can be simple expression or assignment statements, or control-flow statements. Since MATLAB supports many high-level array operations, there are quite a few types of expressions.

3.2 Simplification Process

Although it is possible to define program analyses over the high-level AST, such an analysis must be able to handle arbitrarily complicated expressions and must correctly handle the semantics of MATLAB constructs. Thus, we have defined a lower-level, simpler AST called McLAST. Figure 3 shows the overall simplification process. The front-end delivers the high-level AST (also called McAST) and we wish to create a semantically equivalent lower-level representation.

One of the first surprises for us was that we could not create a lower-level AST before we resolved the meaning of all identifiers. For example, it is not possible to correctly simplify an expression of the form `a(f(end))` before one knows whether `f` is a function or a variable. Thus, before simplification, *kind analysis* [9] must be applied to determine the meaning of identifiers (AST nodes of type Name). The *kind analysis* has itself been implemented using the McSAF framework presented in this paper, and it computes, for each Name node, one of the following kinds: VAR, the Name refers to a *variable*; FN, the Name refers to a *named function*; or ID, the Name could be either a VAR or FN at run-time.

Another challenge is that the simplifying transformations depend on each other, and we wanted to be able to: (1) support applying only some transformations; and (2) allow for new transformations to be added in a consistent and simple fashion. Thus, each simplification forms part of a dependency structure which is enforced by the framework.

In the following sections, Sec. 3.3 to Sec. 3.6 outline the highlights of the individual simplfications, and then in Sec. 3.7 we describe our approach to handling dependencies between simplifications. We also provide some empirical measurements on the frequency of simplifications in Sec. 3.8. More detailed descriptions can be found in the first author's thesis [8] and in the implementation [3].

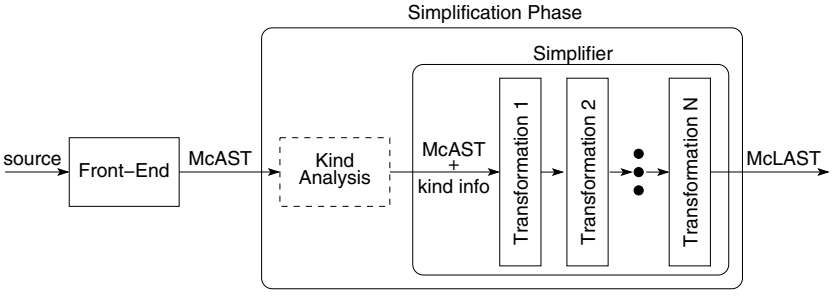


Fig. 3. Simplifications

3.3 Exposing Implicit Control Flow

One important group of simplifications is to expose implicit control flow. Explicit and simple control flow makes the subsequent implementation of flow-sensitive analyses much more straightforward.

The case of MATLAB short-circuit operators illustrates the difficulty in capturing the semantics of MATLAB, which often includes special and somewhat irregular rules that seem to have emerged over time. MATLAB supports implicit control flow via the scalar logical operators `&&` and `||`, which are always short-circuit operators in the usual sense. The logical operator simplification transforms all occurrences of these operators to equivalent explicit control flow with conditional statements, ensuring that code copying is minimized. For example, the original expression in Figure 4(a) would be converted to the conditional statement in Figure 4(b).

```
1 t = E1 && E2;
```

(a) original

```
1 t = E1;
2 if (E1)
3   t = E2;
4 else
5   t = false;
6 end
```

(b) first try

```
1 t = E1;
2 CheckScalarStmt(t);
3 if (E1)
4   t = E2;
5   CheckScalarStmt(t);
6 else
7   t = false;
8 end
```

(c) correct

Fig. 4. Example of simplifying short-circuit `&&`

In implementing the short-circuit transformations we were careful not to duplicate code and we were also quite careful to capture the correct MATLAB semantics of short-circuits. In particular, the MATLAB semantics for the scalar logical operators dictate that the result of any operand that is evaluated must be a scalar logical, and if it is not a scalar, a run-time error is raised. However,

the same run-time check is not made for conditional expressions for `if` and `while` statements. These conditional statements are considered true when the result is non-empty and contains all nonzero elements (logical or real numeric), and false otherwise. Thus, to maintain the correct semantics the simplification introduces new `CheckScalarStmt` nodes, as shown in Figure 4(c).

Another non-obvious twist is that MATLAB also has implicit short-circuits for the element-wise operators `&` and `|`, but **only** when they appear as the top-level operators in the condition of an `if` or `while` construct. Thus, the simplification for element-wise logical operators must ensure that the short-circuit simplification is applied in the correct contexts. In particular, such simplifications must be done before another transformation moves the expression out of the condition.

3.4 Simplifying Control Constructs

In addition to exposing implicit control flow, simplifications are also applied to `if` and `for` statements. The `if` simplification simply restructures `elseif` constructs to equivalent nested `if-else` constructs. This ensures that subsequent flow analyses only have to deal with two control-flow branches.

The `for` simplification is somewhat more MATLAB-specific. Many MATLAB `for` loops are written in the style of Figure 5(a), where `i` takes on values from 1 to `n` in steps of `k`. This style of loop, which we call a *range for loop*, is ideal for subsequent analysis and loop transformations. However, the general form of a MATLAB `for` loop is shown in Figure 5(b). The general semantics is that the expression `E` is evaluated to an array `a`, and `a` is treated as a two-dimensional array. The loop iterates over `a`, assigning to `i` the i'th column of `a`. If `a` is empty, then the loop body does not execute, and the final value of `i` is the empty array. Figure 5(c) shows the simplification that converts a general array to an equivalent range `for` loop.

```
1 for i = 1:k:n
2     <BODY>
3   end
```

(a) range for loop

```
1 for i = E
2     <BODY>
3   end
```

(b) original general loop

```
1  t1=E;
2  [t2,t3] = size(t1);
3  i = [];
4  for t4 = 1:t3
5    i = t1(:,t4);
6    <BODY>
7  end
```

(c) simplified loop

Fig. 5. Example of simplifying `for` loops

3.5 Simplifying Single Statements and Expressions

A key part of the simplification process is simplifying single assignment statements and expressions. The key idea is that each statement and expression is simplified as much as possible, thus reducing the complexity for subsequent analyses.

The first simplification merely divides an assignment so that it has a complex expression only on the right-hand-side (rhs) or only on the left-hand-side (lhs). Thus, statements of the form `E1=E2;` are transformed to `t1=E2; E1=t1;`. Subsequent simplifications then simplify either the rhs (*RValue*) or the lhs (*LValue*).

Each *RValue* is simplified so that it contains at most one complex operation (function call, operand, index, field access or range expression). For example, assuming that `x`, `a` and `i` have kind VAR, the *RValue* expression in `lhs=a(f(g(i),sin(x))).b` would be simplified to `t1=g(i); t2=sin(x); t3=f(t1,t2); t4=a(t3); lhs=t4.b`.

Simplifying complex expressions that are *LValue*'s (i.e. expressions on the lhs of assignment statements) is more difficult because the expression now denotes an address and not a value and MATLAB has no natural way of expressing addresses. Thus, the simplification of *LValue*s simplifies internal expressions as much as possible, but does allow for a chain of indexing and field expressions. The grammar in Figure 6 gives the rules for *LValue*, with a further restriction that NameExpr must refer to names with kind VAR. If we take the same expression as before but use it as an *LValue*, as in the assignment statement `a(f(g(i),sin(x))).b = rhs`, the simplification would be: `t1=g(i); t2=sin(x); t3=(t1,t2); a(t3).b = rhs`. Note that the final statement in the simplification uses the chain of references `a(t3).b`, which cannot be further simplified.

$$
\begin{aligned}
LValue &:= \texttt{NameExpr} \\
&\mid Indexing \\
&\mid Access
\end{aligned}
$$

$$
\begin{aligned}
Indexing &:= \texttt{NameExpr}(NameOrVal^*) \\
&\mid Access(NameOrVal^*)
\end{aligned}
$$

$$
Access := LValue.\texttt{Name}
$$

$$
\begin{aligned}
NameOrVal &:= \texttt{NameExpr} \\
&\mid \texttt{LiteralExpr}
\end{aligned}
$$

Fig. 6. LValue grammar

There is one additional MATLAB-specific expression that must be simplified correctly, which is the `end` expression. The `end` expression binds to the tightest enclosing array or cell array, and it returns the last index of the dimension in which it appeared. For example, the expression `a(foo()).b(i,end,k)` where `a` has kind VAR and `foo` has kind FN, `end` refers to the last index of the second dimension of the 3-dimensional view of the array resulting from the evaluation of `a(foo()).b`. The `end` simplification replaces the `end` expression with an

explicit call to the `endfn` function which has three arguments: the array, the dimension in which the `end` expression was found, and the total number of dimensions. For the example above, the simplification is quite straightforward and would be `t1=foo(); t2=a(t1); t3=t2.b; t4=endfn(t3,2,3); t3(i,t4,k)`. However, more complex situations can arise, especially when `end` is used inside an *LValue*. Consider the example, `a(foo()).b(i,bar(end),k)=rhs`, assuming `bar` has kind Fn. This will be simplified to `t1=foo(); t2=endfn(?,2,3); t3=bar(t2); a(t1).b(i,t3,k)=rhs` where the `?` must be filled in with correct simplified *LValue*, which in this case is `a(t1).b`.

3.6 Dealing with Multiple Assignments

In the previous simplifications we have assumed that assignments have a simple lhs. However, MATLAB supports assignments to multiple variables on the lhs. Dealing with multiple values on the lhs is not usually difficult, but with MATLAB there is an important complication. That is, that MATLAB allows an unknown number of lhs variables. The simplifications handle the simple and more complicated cases as follows.

In the case where the number of lhs variables is known, the simplification ensures that only simple variables, without repetition occur on the lhs. For example, `[a,b.c,a]=lhs` would be simplified to `[a,t1,t2]=lhs; b.c=t1; a=t2`. This simplification simplifies subsequent interprocedural analyses since the number and names of the return parameters are explicit.

The case where the number of variables on the lhs is not known is harder, and it was quite difficult to decide how to handle this case. In the end we decided to introduce a new `CSL` IR node. A typical example is the statement `[a,b{:},c]=rhs`. In MATLAB this specifies that the first return value is bound to `a`, the last return value is bound to `c`, and the middle values are bound to the cell array `b`. The simplification for these cases introduces an explicit `CSL` node for each item in the return list than can possibly correspond to a list of values (known in MATLAB as a *Comma Separated List* (CSL)). For each such CSL, the simplification introduces an explicit `CSL` node associated with a new temporary name. For example, `[a,b{:},c]=rhs` would be simplified to `[a,CSL[t1],c] = rhs; [b{:}] = CSL[t1]`.

3.7 Simplification Dependencies

The complete simplification procedure is implemented as a collection of simplifying transformations. Some simplifications depend upon others having already been applied. The dependencies between the existing simplifications is given in Figure 7. The simplification called `FULL` represents the case where all simplifications should be applied. However, framework users might want to only apply some simplifications, for example loop transformations may just want to apply the `FOR` simplifications.

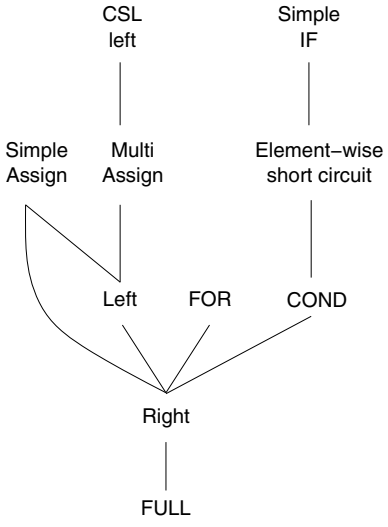


Fig. 7. Dependencies for Simplifications

To enforce the dependencies, the framework provides the Simplifier class. In addition, each simplification is implemented as a class extending AbstractSimplification. The AbstractSimplification class requires that each simplification have a method called getDependencies that returns a set of dependencies. In order to use the simplifier, an instance must be constructed with a given set of simplifications to perform. The simplifier then performs a depth first traversal of the dependency DAG producing a list of simplifications, avoiding duplication. Executing the simplifications in the order of the list will ensure that all dependencies will be met. To make it simpler to perform any given simplification and its dependencies, each simplification has a getStartSet static method. This method returns a singleton set containing the simplification itself. Clearly framework users can add new simplifications and state their dependencies quite easily. In fact, the MATLAB Taming project has recently built upon the McSAF framework by introducing new simplifications which were inserted quite easily using this approach.

3.8 Simplification Frequencies

In order to examine the relative frequency applications for each simplification type, we instrumented our simplification framework to count the number of times each simplification caused a statement to be rewritten and the number of times the simplification extracted an expression to a temporary variable. We applied the instrumented simplifier to a large collection of MATLAB functions

and scripts.[3] The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing. We analyzed 3057 projects composed of 11698 functions and 2349 scripts. The projects vary in size between 283 files in one project to a single file. A summary of the data collected is given in Figure 8, ordered by increasing frequency.

Simplification	# rewrites	(%total)	# temps extracted
FOR	329	(0.1%)	329
Multi-Assign	354	(0.1%)	651
Element-wise short-circuit	1791	(0.5%)	4068
Simple IF	4518	(1.3%)	0
Left	6649	(1.9%)	7969
Simple-Assign	24478	(7.1%)	24478
Right	306397	(88.9%)	325780

Fig. 8. Frequencies for Simplifications

These results show that the FOR and Simple IF transformations are applied relatively infrequently. The benchmarks contained 12189 `for` statements and only 329 required the FOR simplifications. Similarly only 4518 `elseif` statements needed to be simplified, out of a total of 31758 `if` statements in the benchmarks. We were somewhat surprised that there were almost 1800 occurrences of the element-wise short-circuit simplification. The use of element-wise short-circuiting is discouraged by MathWorks, but it appears that existing code does use this feature. This may be a potential refactoring opportunity. It was also interesting to note that there were relatively few applications of the Multi-Assign simplification, especially as compared to the Simple-Assign. As expected, by far the most frequent simplification was the Right simplification which simplifies expressions by extracting sub-expressions to a temporary.

4 Analysis Framework

A key part of the McSaf toolkit is an analysis framework that supports a variety of pre-defined and extensible traversal mechanisms; built-in and extensible support for representing a variety of flow data types; and support for depth-first and structure-based analyses. The toolkit has been designed to work both for the higher-level AST, as well as for the lower-level McLast IR.

[3] Benchmarks were obtained from individual contributors plus projects from `http://www.mathworks.com/matlabcentral/fileexchange`, `http://people.sc.fsu.edu/~jburkardt/m_src/m_src.html`, `http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/` and `http://www.mathtools.net/MATLAB/`. This is the same set of benchmarks that are used in [9].

4.1 Traversal Mechanism

The traversal mechanism is central to the framework - in fact it is used both to drive the simplifications presented in the previous section and the analyses presented later in this section. The framework accommodates different traversals by implementing a variant of the visitor pattern. The IR consists of instances of different types of AST nodes. The types form a class hierarchy, which is induced by the JastAdd specification, an excerpt of the hierarchy is depicted in Figure 9.

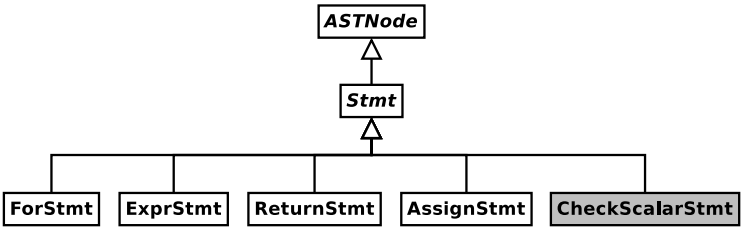


Fig. 9. Excerpt of AST class hierarchy. The grey class, CheckScalarStmt is an AST node that is part of McLast and not McAst. All white classes in this diagram are part of both McLast and McAst.

To facilitate traversal, there is a Java interface, NodeCaseHandler, that consists of methods of the form void caseStmt(Stmt node). There is one such method for every AST class. The framework also provides a simple abstract implementation called AbstractNodeCaseHandler. This implementation provides default behaviour for each node case. This default behaviour is that for each AST class, the node case for that class simply forwards to the node case of its parent class. The forwarding is done by calling the case for the parent class with the input from the case for the given class. We demonstrate this with the following excerpt from AbstractNodeCaseHandler for the AssignStmt and Stmt classes. Notice that case AssignStmt(...) is forwarding up to the case belonging to its parent class, Stmt.

```
public void caseAssignStmt( AssignStmt node ) { caseStmt( node ); }
public void caseStmt( Stmt node ) { caseASTNode( node ); }
```

Figure 9 shows that the AssignStmt node type extends the Stmt node type. This means that the default behaviour for case AssignStmt(...) is to forward to case Stmt(...), which is done by passing the argument from case AssignStmt(...) to case Stmt(...). The definition for the case AssignStmt(...) method demonstrates the forwarding behaviour. This method takes in an instance of AssignStmt and calls case Stmt(...) with that instance. Note that ASTNode is the root type of the AST class hierarchy. The Stmt class is a top level node type, which directly extends ASTNode, so the case Stmt(...) will forward to case ASTNode(...). The AbstractNodeCaseHandler leaves the case ASTNode(...) method unimplemented.

Each AST class implements a method called analyze that takes a NodeCaseHandler as an argument. These methods will call the appropriate node case of the given handler, passing itself to the handler. For example, here is the code implementing the analyze method in the AssignStmt class.

```
public void analyze( NodeCaseHandler handler ) { handler.caseAssignStmt( this ); }
```

In order to create a particular traversal, a programmer needs to create a specialized NodeCaseHandler. The different types of analyses are implemented in this manner, but a programmer can also directly create a specialized traversal. To demonstrate this process we present a simple traversal, called StmtCounter, that counts the number of statements in a given AST. Code for this traversal is given is Figure 10.

```
1  public class StmtCounter extends AbstractNodeCaseHandler {
2    private int count = 0;
3    private StmtCounter() { super(); }
4
5    public static int countStmts( ASTNode tree ) { tree.analyze( new StmtCounter() ); }
6
7    public void caseASTNode( ASTNode node )
8      { for( int i = 0; i<node.getNumChild(); i++ )
9          node.getChild(i).analyze( this );
10     }
11
12   public void caseStmt( Stmt node ) { count++; caseASTNode( node ); }
13 }
```

Fig. 10. Example traversal counting statements

To use this class, a programmer simply needs to call the static method countStmts. This method creates a new instance of the traversal and starts the analysis off.

This traversal will visit all nodes in the tree in depth-first order, and count each statement node. There are two important details to note from this example. The first thing is the case ASTNode(...) implementation. In this example, this method does the actual traversal over the tree, looping through and visiting each of a node's children. Since StmtCounter extends AbstractNodeCaseHandler all cases that are not overridden will forward up until they reach this case. This means that the default behaviour for AST nodes will be to simply traverse through their children. This is a common pattern when implementing traversals. The flow-insensitive traversal is implemented similar to this, and the flow-sensitive traversals have a similar case ASTNode(...) with other behaviour implemented for control structures like loops and conditionals.

The second thing to notice is the case Stmt(...) method. Besides case ASTNode (...), this is the only case implemented by StmtCounter. Again, since StmtCounter extends AbstractNodeCaseHandler, all node types that are descendants of Stmt

will forward up to this case. So this case will capture all statements, which gives a good place to perform the count. Note that this implementation of case Stmt-(...) forwards to case ASTNode(...). This is because there are some statements, such as if statements, that can contain other statements. We wish to visit all of the statements contained in other statements, so we need to visit the children of a given statement. To do this, we simply forward to case ASTNode(...).

The StmtCounter example does have some inefficiencies. It will visit all children of a given node, even children that cannot be, or contain, statements. For example, the children of an expression cannot be a statement, nor can it contain statements. This shortcoming can be overcome by providing specialized implementations of appropriate cases. To avoid visiting unnecessary expression children one could add the following method to the class. This method will prevent all children of any expression from being visited.

```
...
public void caseExpr( Expr node ) { return; }
...
```

The example can be refined further, but the original version is concise and correct, and demonstrates how simple it can be to create new traversals. The traversal mechanism is also used by the simplifications presented in Sec. 3. There is a specialized traversal created for all simplifications. This traversal implements the rewrite mechanisms as well as the AST traversal. Each simplification extends this simplification traversal, implementing it's own behaviour for the appropriate node cases.

4.2 Representations for Flow-Data

An analysis is written to produce information about the program being analyzed. McSaf's analysis classes are generic in the type of information produced. An analysis of type StructuralAnalysis <D> is an analysis that produces information of type D. To make the framework as general as possible, the information can be of any type. However, the type of information often falls into certain categories. One example is an analysis that produces, for every program point, a set of variables that must be defined at that program point. Alternatively, for every program point, the analysis could have produced a map from variable names to their types. To make implementing analyses that produce such information easier, the McSaf framework defines interfaces and implementations for basic flow-data. A class hierarchy for flow-data structures provided by the framework is given in Figure 11.

The FlowData<D> interface is the base type for all predefined flow-data representations. This type represents a collection of data of type D. The interface is primarily intended to tag a class as representing flow-data. As such, it defines no methods. In addition to this basic interface, the framework also provides two more specific interfaces, one for sets and the other for maps. For each of these,

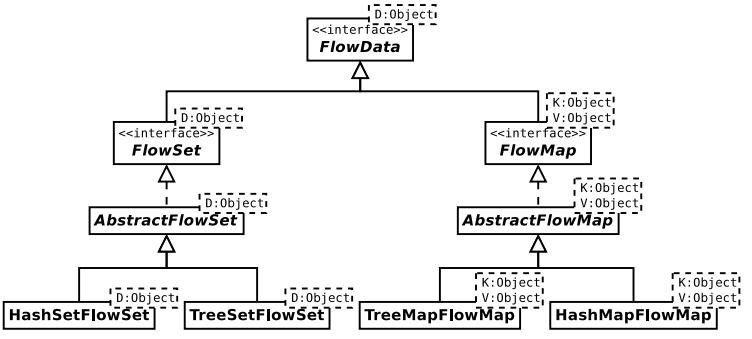


Fig. 11. flow-data class hierarchy

an abstract implementation is provided to make creating new implementations simpler. In addition, each of these interfaces also have concrete implementations.

4.3 Depth-First Analysis

The simplest form of analysis supported by the framework is the depth-first analysis. This type of analysis is intended to traverse the tree structure of the AST, visiting each node in a depth-first order. The depth-first analysis type can be used to implement flow-insensitive analyses.

This type of analysis is implemented by extending the AbstractDepthFirstAnalysis<A> class. The AbstractDepthFirstAnalysis implements the Analysis interface and extends AbstractNodeCaseHandler. This relationship is depicted in Figure 12.

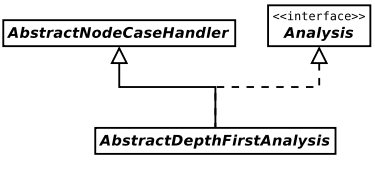


Fig. 12. Class hierarchy snipet for depth-first analysis

Since AbstractDepthFirstAnalysis extends AbstractNodeCaseHandler it inherits all the default traversal behaviour. It extends this behaviour with default implementations of the new case methods defined by the Analysis interface. The behaviour for these new cases is to forward to the case associated with the type of the argument that the case accepts. For instance case LoopVar(AssignStmt loopVar) accepts an AssignStmt. So the default behaviour will be to forward to case AssignStmt(...). The case WhileCondition(...) and case IfCondition(...) are

exceptions to this. These cases are specialized versions of case Condition(...) so they will both forward to case Condition(...) by default.

The most important feature of AbstractDepthFirstAnalysis is that it implements a case ASTNode(...) method. The implementation of this method provides the basic traversal for this type of analysis. Figure 13 presents the source code for this method. The case ASTNode(...) method takes in the ASTNode being visited. For each child of that node, that child is analyzed. So to reiterate, since AbstractDepthFirstAnalysis extends AbstractNodeCaseHandler, and due to its implementation of case ASTNode(...), the default behaviour for every node is to simply analyze all children of that node.

```
1 public void caseASTNode(ASTNode node)
2 { //visit each child node in forward order
3   for( int i = 0; i<node.getNumChild(); i++ ){
4     if( node.getChild(i) != null )
5       node.getChild(i).analyze( callback );
6     }
7 }
```

Fig. 13. Depth-first caseASTNode(...) source code

AbstractDepthFirstAnalysis also defines some new methods and fields for storing and accessing the data being produced by the analysis. It provides a map from AST nodes to the data being computed. This allows data to be associated with any desired node. In order to implement a new depth-first analysis, a programmer must create a concrete class that extends AbstractDepthFirstAnalysis. To create this class, a programmer must: (1) select an analysis data type; (2) implement an appropriate newInitialFlow method; and (3) implement an appropriate constructor. This will result in an analysis that will traverse the entire tree visiting each node in depth-first order. To get the analysis to perform a useful task, the programmer must override appropriate case methods. The analysis will usually build up flow-data, and can also associate flow-data with particular nodes in the tree.

To demonstrate the process of implementing a depth-first analysis, we present a simple example analysis, given in Figure 14. This analysis collects all names that are assigned to. It performs two tasks. First, for each assignment statement in the tree, it associates all names that are assigned to by that assignment statement to the assignment statement. Second, it collects in one set, all names that are assigned to in the entire AST. Names are stored as strings, so the flow-data has type HashSetFlowSet<String>. The analysis defines a field to store the full set of names, and a flag for indicating when the traversal is in the lhs of a statement. There are two accessor methods, one to get all the names, and the other to get the full set. The core of the analysis is defined by the three "case" methods which guide the traversal and collect the information.

```
1  public class NameCollector extends
2              AbstractDepthFirstAnalysis<HashSetFlowSet<String>>
3  { private HashSetFlowSet<String> fullSet;
4    private boolean inLHS = false;
5
6    public NameCollector(ASTNode tree)
7    { super(tree);
8      fullSet  = new HashSetFlowSet<String>();
9    }
10
11   public HashSetFlowSet<String> newInitialFlow()
12   { return new HashSetFlowSet<String>(); }
13
14   public Set<String> getAllNames() { return fullSet.getSet(); }
15
16   public Set<String> getNames( AssignStmt node )
17   { HashSetFlowSet<String> set = flowSets.get(node);
18     if( set == null ) return null; else return set.getSet();
19   }
20
21   public void caseName( Name node )
22   { if( inLHS ) currentSet.add( node.getID() ); }
23
24   public void caseAssignStmt( AssignStmt node )
25   { inLHS = true;
26     currentSet = newInitialFlow(); // init set for this stmt
27     analyze( node.getLHS() ); // analyze only the lhs
28     flowSets.put(node,currentSet); // store names in node
29     fullSet.addAll( currentSet ); // add to full set
30     inLHS = false;
31   }
32
33   public void caseParameterizedExpr( ParameterizedExpr node )
34   { analyze(node.getTarget()); } // only the target can be a defn
35 }
```

Fig. 14. NameCollector definition

4.4 Structure-Based Analysis

Structure-based flow analysis is the core part of the analysis framework. Structure-based flow analyses perform a complete forwards or backwards analysis over the AST or McLast representation, merging control flow for conditional and switch statements and computing fixed-points for loops, including the proper handling of `break` and `continue` statements. The computational part is driven via specialized traversal mechanisms, either forwards or backwards. The user of the framework only needs to focus on the analysis rules for basic statements and the

implementation of the flow-data type (which is either a direct use of a flow-data type provided by the framework or a specialized user-provided type).

The organization of the structure-based analyses are illustrated in Figure 15. The abstract implementation, called AbstractStructuralAnalysis, provides constructors and implementations for most of the API methods and has extensions to support forwards and backwards analyses.

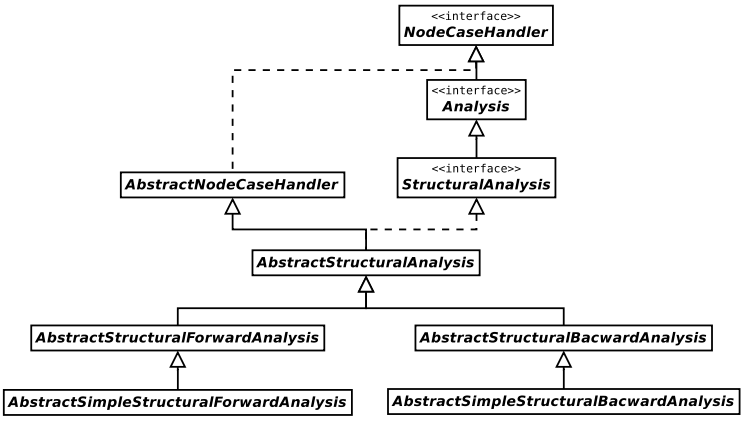


Fig. 15. Class hierarchy snipet for structural analysis

The AbstractStructuralAnalysis implementation is similar to the AbstractDepthFirstAnalysis implementation in that it also provides a protected method void analyze(ASTNode node) and is also intended to abstract away from the basic traversal mechanism. Unlike AbstractDepthFirstAnalysis, AbstractStructuralAnalysis does not provide an implementation for case ASTNode(...). This is because structural analyses are split into two flavours, forwards and backwards, and each of these flavours requires its own implementation of case ASTNode(...). The forwards and backwards analyses are implemented in a similar way. For each, the framework provides a general abstract implementation and a simple abstract implementation.

The general implementation provides an implementation for the basic API methods. It also provides an implementation for some traversal methods, including loops and conditionals. These implementations for the traversal methods are what makes analyses derived from these classes capable of computing flow-sensitive analyses. In the case of the loop cases, case ForStmt(...) and case WhileStmt(...), they provide the fixed-point computation procedure, and the traversal also correctly handles the control-flow due to `break` and `continue` statements.

The simple implementations go beyond this core functionality. They implement certain behaviour that would not be applicable to all analyses. Such behaviour includes how to deal with `continue` and `break` statements. These implementations represent the functionality needed to write analyses that do not

need more complex behaviour. They were provided to make analyses simpler to write, requiring less duplication of code.

Handling Control Flow: Unlike analysis frameworks that operate on control flow graphs, our framework computes the flow information based on the structure of the AST. Figure 16 gives a high-level diagrammatic view of how the traverser handles the control flow. Figure 16(a) demonstrates that the condition is evaluated first, and then the flow information is sent to the `if` and `else` branches. The user of the toolkit is given the ability to specialize the flow sets for each branch, so an analysis can use the results of analyzing the condition to specialize the analysis for different branches. The $\bowtie$ operator demonstrates where the data flow merge operation is applied. Figure 16(b) gives the traversal for switches which demonstrates that for MATLAB the conditions can be arbitrary expressions that must be evaluated before the body of the case. Figure 16(c) and (d) illustrate the behaviour for `while` and `for` loops. In this case the framework takes care of merging the flow information from breaks and continues at the appropriate places, as well as computing a fixed point.

These general implementations represent the core functionality that is needed for these types of analyses. This functionality should be applicable to most analyses of this type, and most flow analysis developers should not have to override them. However, should a flow analyses developer have a special kind of fixed point that he/she would like to implement, this can be done by specifying a new specialized traversal.

Creating an Analysis Instance: To create a forwards analysis, a programmer must extend one of the forward classes from Figure 15. The flow-analysis computations are implemented in the case methods for various node types. The caseASTNode(...) implements the basic traversal. It does this by looping through the children of a given node and using the provided analyze(ASTNode node) method. Recall that this method deals with basic traversal and also guarantees that the current InSet is set to the previous current OutSet. The case IfStmt(...) and caseSwitchStmt(...) implement the behaviour for non-looping branching code. The `if` statement behaviour provides what we call branching analysis. This means that, if the analysis writer wishes, they can provide a different out flow for when the if condition is true or false. This would be done in an implementation of case IfCondition (...). When a programmer provides true and false flow-data, case IfStmt(...) will ensure that each branch of the `if` will have the appropriate in flow-data.

Creating a backwards analysis is very similar, except that the backwards analysis does not support the option of different flow information for the different branches of an `if`.

4.5 Analyses for Language Extensions

One of the goals of the McLab project is to support modularly defined language extensions (as illustrated by the extension for AspectMatlab [20]). Extensibility is also one of the goals for the design of McSaf. The framework has been

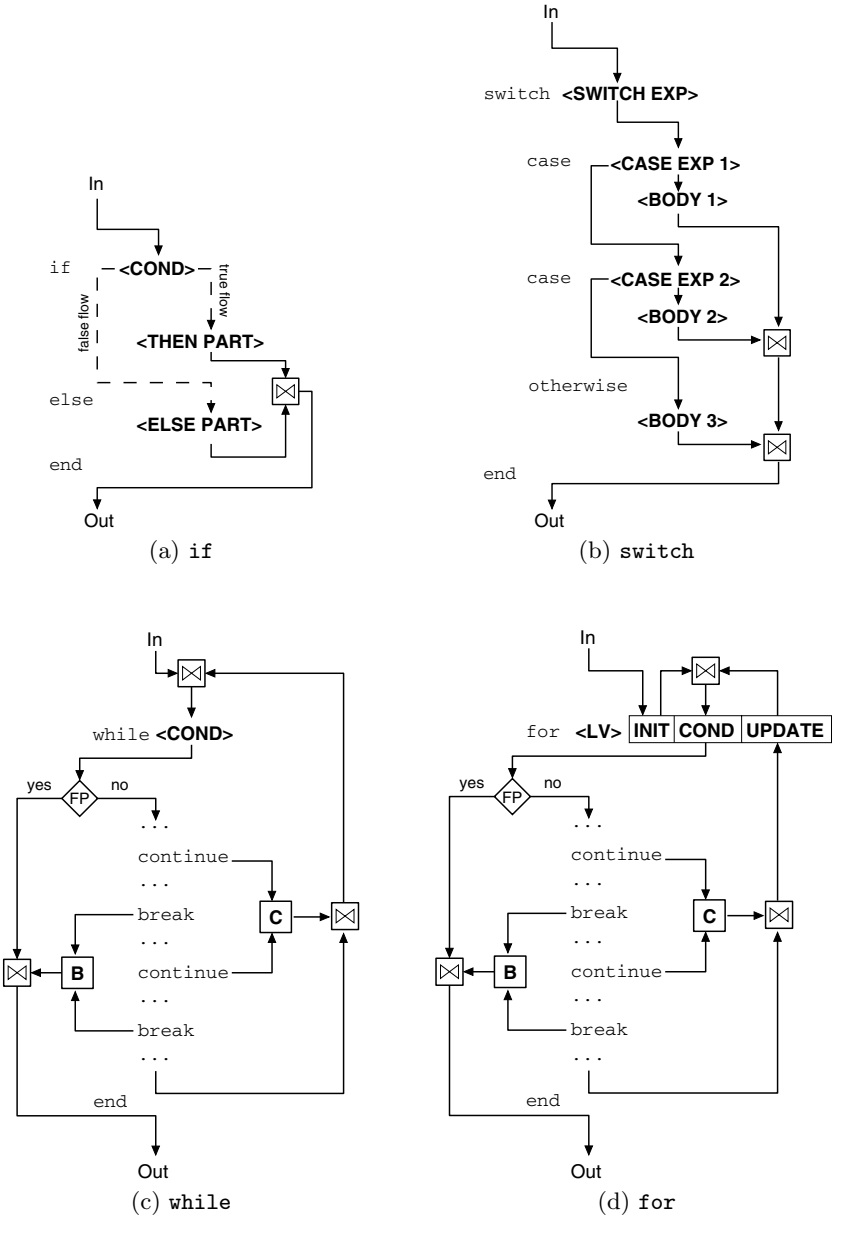


Fig. 16. Diagrammatic illustration of structural traversal rules

designed to support three kinds of extensions: (1) adding new syntactic nodes that are desugared before analysis; (2) adding nodes that need to be included in existing and new analyses; and (3) adding nodes that require new kinds of complex flow analysis - for example a new kind of loop. The framework comes

with three example language extensions, one for each type. Although the second and third kind require slightly more glue code, all three kinds of extensions can be accomplished by defining new pieces and reusing the previous code.

5 Related Work

The McLab Static Analysis Framework is an extensible framework for creating static analyses for the MATLAB language. This is the first open framework created for analyzing MATLAB.

Soot [21] is one example of an optimization framework for Java which was developed by our research group for over a decade. McSaf emulates many of the positive features of Soot, including having a well-defined IR and a flexible and easy-to-use flow analysis framework. However, McLab solves very MATLAB-specific problems in the IR design and uses a structured traversal-based approach (rather than a graph-based approach) to the flow analysis framework. So, in the end we found our experience with Soot helped us know the goals for the McSaf project, but developing a framework for MATLAB required solving very different problems. Soot has enabled a lot of research for Java and we hope that McSaf will do the same for MATLAB.

The JastAdd toolkit is designed for creating extensible compilers [2,12] and was also used in the development of McLab and McSaf. One feature of JastAdd that was not discussed in great detail in this paper is its attribute grammar system. JastAdd allows a developer to define attributes as part of the AST grammar specification. These attributes are effectively functions operating on the AST nodes. They can be used to propagate information through the tree and they can even be defined in a circular fashion. The JastAdd system provides a fixed-point computation for calculating the results of such circular attributes. JastAdd's attribute system provides a low-level means of performing flow analysis on an AST. It is up to the compiler writer to use these tools and to take the semantics of the language they are implementing into account, in order to create any meaningful analyses. It isn't a full dataflow analysis framework. However, some work [17] has been done to implement flow analysis for Java using the JastAdd extensible Java compiler [11].

In the past, there has also been some work towards open MATLAB systems such as Octave [1]. These systems concentrate on front-ends and interpreters and so do not include analysis and optimization frameworks.

There has also been work on optimizing MATLAB. The FALCON project [19,7] aimed to compile MATLAB code into FORTRAN. Falcon focuses on type inference and code inlining to produce FORTRAN code. The Magica tool [14,13] focuses on type inference for matrix operations and functions. It not only infers the intrinsic type of matrices, such as `int32`, `double`, or `char`; but also matrix sizes and shapes. Magica is part of a larger MATLAB compiler project, and is used for performing code optimizations. MaJIC incorporates a Just-In-Time(JIT) compiler component [4]. This allows it to achieve speedups similar to those produced by Falcon, without sacrificing the interactive nature of MATLAB. These projects

differ from McSaf in that their main goal was to improve the performance of Matlab programs. McSaf, on the other hand, was created with the goal of creating an open tool for researching compiler techniques in scientific programming. In fact the techniques used in these other projects could have been implemented using McSaf.

6 Conclusions and Future Work

In this paper we have presented the McSaf framework, an open source toolkit for developing analyses for Matlab. The goal of the toolkit is to enable compiler researchers to develop a wide variety of analyses which can target optimizations, source-to-source translators, and software engineering tools such as refactoring tools.

The toolkit includes a suite of simplifications which convert the high-level AST to a lower-level AST which is designed to expose important Matlab-specific semantics and to provide a simple IR which eases the burden of developing new analyses rules. Our simplifying transformations have been implemented along with a dependency structure so that it is easy for a user to enable only some simplifications or add new simpflications, while at the same time ensuring that all prerequisite simplifications have already been performed.

The heart of the toolkit is the support for different kinds of flow analysis driven by a collection of traversals, including a depth-first traversal which works well for flow-insensitive analyses and a family of structure-based traversals for forward and backward flow-sensitive analyses.

Developing this toolkit was not just an engineering exercise. At each step we had to understand the very Matlab-specific requirements and ensure that our approach captured the correct semantics in a way that made the IR and flow analysis framework easy to use. Indeed, we found the entire exercise much harder than we had anticipated, with the Matlab semantics often going against our expectations or enforcing some constraint that was not obvious.

Our goal was to make a toolkit that is easy to understand, and which is easy to extend. We don't want the compiler/analysis/tool developer to have to worry about all of the details of Matlab, but rather concentrate on using a well-structured object-oriented toolkit that provides an IR and analysis framework which has dealt with the messy details.

The McSaf toolkit has been implemented in Java as part of the McLab system and numerous analyses have already been implemented using it, including those mentioned in Figure 1. To date we have found that toolkit users find it quite easy to use and we look forward to feedback from users to help us improve it further.

Our intention is to continue using the toolkit in our own back-ends and tools, and we hope that other compiler researchers will also be able to use the toolkit as a simple and low overhead way to apply their research to the very popular Matlab language.

References

1. GNU Octave, http://www.gnu.org/software/octave/index.html
2. JastAdd, http://jastadd.org/
3. McLAB (2011), http://www.sable.mcgill.ca/mclab/
4. Almási, G., Padua, D.: MaJIC: compiling MATLAB for speed and responsiveness. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 294–303. ACM, New York (2002)
5. Casey, A., Hendren, L.: MetaLexer: A modular lexical specification language. In: AOSD, pp. 7–18 (2011)
6. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through Just-In-Time Specialization. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 46–65. Springer, Heidelberg (2010)
7. Derose, L., De Rose, L., Gallivan, K., Gallivan, K., Gallopoulos, E., Gallopoulos, E., Marsolf, B., Marsolf, B., Padua, D., Padua, D.: FALCON: A MATLAB Interactive Restructuring Compiler. In: Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 269–288. Springer, Heidelberg (1996)
8. Doherty, J.: McSAF: An extensible static analysis framework for the MATLAB language. Master's thesis. McGill University (December 2011)
9. Doherty, J., Hendren, L., Radpour, S.: Kind analysis for MATLAB. In: OOPSLA (2011)
10. Dubrau, A.: Taming MATLAB. Master's thesis. McGill University (April 2012)
11. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, pp. 1–18. ACM, New York (2007)
12. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Science of Computer Programming 69(1-3), 14–26 (2007)
13. Joisha, P.G.: A Type Inferenence System for MATLAB with Applications to Code Optimization. Ph.D. thesis, Northwestern University (2003)
14. Joisha, P.G., Banerjee, P.: The MAGICA Type Inference Engine for MATLAB. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 121–125. Springer, Heidelberg (2003)
15. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: CGO, pp. 75–88 (2004)
16. Li, J.: McFor: A MATLAB to FORTRAN 95 compiler. Master's thesis. McGill University (August 2009)
17. Nilsson-Nyman, E., Hedin, G., Magnusson, E., Ekman, T.: Declarative intraprocedural flow analysis of Java source code. Electron. Notes Theor. Comput. Sci. 238, 155–171 (2009)
18. Radpour, S.: Understanding and Refactoring MATLAB. Master's thesis, McGill University (April 2012)
19. Rose, L.D., Padua, D.: Techniques for the translation of MATLAB programs into Fortran 90. ACM Trans. Program. Lang. Syst. 21(2), 286–323 (1999)
20. Toheed Aslam, A.D., Doherty, J., Hendren, L.: AspectMatlab: An aspect-oriented scientific programming language. In: AOSD, pp. 181–192 (March 2010)
21. Vallée-Rai, R., Gagnon, E.M., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)

Multiple Aggregate Entry Points for Ownership Types*

Johan Östlund and Tobias Wrigstad

Uppsala University

Abstract. Deep ownership types gives a strong notion of aggregate by enforcing the so-called *owners-as-dominators* property: every path from a system root to an object must pass through its owner. Consequently, encapsulated aggregates must have a single *bridge object* that mediates all external interaction with its internal objects.

In this paper, we present an extension of deep ownership that relaxes the single bridge object constraint and allows several bridge objects to collectively define an aggregate with a shared representation. We call such bridge objects *ombudsmen* to emphasise their benevolent nature; ombudsmen-sharing is explicit and all ombudsmen are created internal to the aggregate, purposely.

The resulting system brings the aggregate notion close to the component notion found in e.g., UML by clearly separating aggregation from the stronger composition, and further allows expressing common programming patterns such as iterators without resorting to systems that give unclear or unprincipled guarantees, or require additional complex machinery such as read-only references.

1 Introduction

Ownership types allow programmers to express encapsulation properties of programs in a compile-time checkable way. Ownership-based encapsulation has been used in many areas, including verification [22,27,31], reasoning about computational effects [13,34,15], information flow [3], memory management [9], object upgrades [8] and concurrent programming [6,17,15,39].

In classical "Clarkean" ownership types [12] every object belongs to another object and may be the owner of other objects. An *owned* object is said to belong to the *representation* of its owning object and cannot escape outside its owner. Consequently, an external object can only interact with a representation object via the public interface of its owner, which allows maintaining invariants and facilitates automated and manual reasoning. If we think of a heap as a graph of objects, all paths from the root of the graph to an object will include its owner—this is the *owners-as-dominators* property first formulated by Clarke et al. [16].

By imposing a hierarchical structure on the heap, owners-as-dominators gives strong and useful encapsulation guarantees, but at the cost of excluding some

* Supported in part by the Swedish Research Council within the UPMARC Linnaeus centre of Excellence.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 156–180, 2012.

common programming patterns. A common example of such a pattern is the iterator pattern for linked lists, which is the canonical example when introducing ownership types. A linked list should encapsulate its links, but for an iterator object to be able to access the next element in $O(1)$, it requires a direct reference to the "current" link in the list, which is only allowed in ownership systems if the iterator is *internal to the list itself.*

An alternative implementation that does not break the strong encapsulation of owners-as-dominators is to *unencapsulate* the list's links so that they become external to the list and therefore can be referenced by the iterator. This of course destroys the encapsulation.

In a nutshell: either the iterator can only be used inside the list (which renders it useless) or the list's encapsulation must be broken.

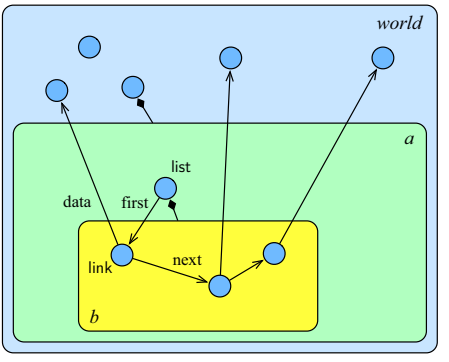
We can interpret a list as a software component with multiple service ports—one which implements the list interface and one which allows iterating over it. In this mindset, it makes sense to think of the links as encapsulated inside the (composite) component instead of inside some particular object that constructs it. However, implementing such a component in a Clarkean ownership system leads to exactly the above-mentioned problem of losing the encapsulation of the links: *there is no way to specify a set of objects shared between two or more objects that collaborate in defining a larger unit.* The problem is the unification of units of encapsulation and objects—the only way to introduce a unit K of encapsulation is by introducing a new object, which by virtue of owners-as-dominators blocks all direct access to the objects in K from external objects.

Contributions. This paper contributes to the field of Clarkean ownership systems by distinguishing between *composition* and *aggregation.* Just like traditional ownership systems, an object can be *composed* from representation objects which it dominates; additionally, an object may also *aggregate* other objects to which it may *share ownership* with other objects in a novel unit of encapsulation called an *aggregate.* Consequently, we can allow multiple entry points into an aggregate.

Our design allows programming idioms which rely on "principled sharing of representation" to be encoded in Clarkean ownership systems without compromising encapsulation by using aggregation instead of composition. The result encapsulation property is as easy to understand and almost as powerful as owners-as-dominators. Concretely, we:

- extend ownership types to support multiple entry points to a shared aggregate in a disciplined way with a strong encapsulation invariant called *ombudsmen-as-dominators*, which is clearly visible in the types;
- provide a simple and intuitive extension, adding only two new keywords;
- design the extension to be "pluggable"—it concerns only objects inside a shared aggregate, the semantics of old keywords is unmodified, and other objects enjoy strong encapsulation with owners-as-dominators;
- formalize the extension in a core language, and prove type soundness and our novel encapsulation invariant;
- have implemented our system on top of a Joline-like type checker; and
- provide an extensive coverage of related work.

```
class List<owner, data> {
  Link<this, data> first;
  void insert(Object<data> e) {
    Link<this, data> l =
                new Link<this, data>();
    l.data = e;
    l.next = first;
    first = l;
  }
}
class Link<owner, data> {
  Object<data> data;
  Link<owner, data> next;
}
```



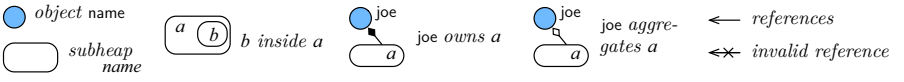


Fig. 1. *Left*: A list with ownership annotations. *Right*: Ownership structure of the linked list. The context a contains the linked list object which defines the context b for its representation objects (its links). Each link has a data field which points to its element objects. In this particular instance, the element objects reside in the outermost context `world`. The type of the list object is therefore `List<a,world>` binding the class' owner parameters `owner` and `data` to `a` and `world`, respectively.

2 Ombudsmen-as-Dominators

In this section we give an informal presentation of our system. First, however, we recap Clarkean ownership types as our encapsulation property *ombudsmen-as-dominators* builds directly on the classic notion of deep ownership types.

We use two main examples: iterators, with the dual purpose of introducing key concepts, and a shared bank account.

2.1 Clarkean Ownership Types

In Clarkean systems, the heap is hierarchically divided into a number of *contexts*, which can simply be thought of as sets of objects. Every object introduces a new context to hold its representation, nested inside the context where the object resides. In the source code, labels that denote run-time contexts are called *owners*. Owners are embedded in types to capture ownership and access permissions.

Linked Lists. Consider the linked list implementation shown in the left of Figure 1. Classes are parametrized over permissions to reference contexts; we call these *owner parameters*. The first owner parameter is always called `owner` and is special in that it also denotes the owner of the list instance, *i.e.,* the context where the list resides. The second parameter, called `data` in this example, is a necessary permission to reference the elements stored in the list.

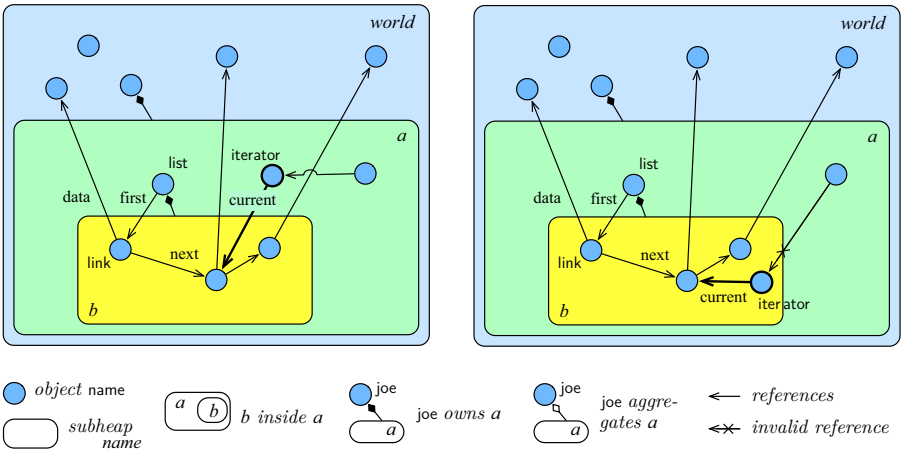


Fig. 2. Linked list with an iterator (showed with thicker lines). The iterator needs to reference the list's links, which breaks encapsulation. On the right, the iterator is moved inside the list shifting the problem to accessing the iterator.

The ubiquitous, implicitly declared owner **`this`** denotes the context introduced by the current object to hold its representation. In the list, all links are owned by **`this`** and are therefore encapsulated inside the list and cannot be exported outside. The `data` owner is forwarded to the links, as they too need permission to reference the element objects. In the `Link` class the `next` field has type `Link<`**`owner`**`,data>`, where **`owner`** is the list's representation.

The right hand side of Figure 1 depicts an instance of a heap with a list and a few contexts denoted by rounded boxes. The box b is the list's representation context, containing all its links, as they are owned by **`this`** in the `List` class. The list elements belong to the context **`world`**, which is the outermost context.

Owners-as-dominators allows references going outwards in the hierarchy, *e.g.,* the links may reference the elements, but not the converse. If an object x references an object y in a context k dominated by object z, then either $x = z$ or x must be inside k. In terms of Figure 1, if a reference crosses *into* a context, then the origin of the reference must be the object that owns the context.

The list implementation in Figure 1 makes sense from an object-oriented design point of view. The links are an implementation detail of the list and should not be observable from the outside. The problem with single entry point aggregates surfaces when we try to add an iterator to the `List` class, depicted in Figure 2 (left). Owners-as-dominators allows outward-going references, but the iterator needs to point inwards, into the list. The only way we can allow the iterator to reference the links is if we move the iterator into the list (context b), but then the iterator cannot be exported to a client of the list, Figure 2 (right).

The general problem is that ownership types cannot express two objects encapsulating a common context, for reasons made clear in the upcoming example.

```
class Person<owner, q> {
  Account<q> account; // private
  Person<owner, q> spouse;
  void share() {
    account = spouse.getAccount();
  }
  Account<q> getAccount() {
    return account;
  }
  ... // omitted
}
```

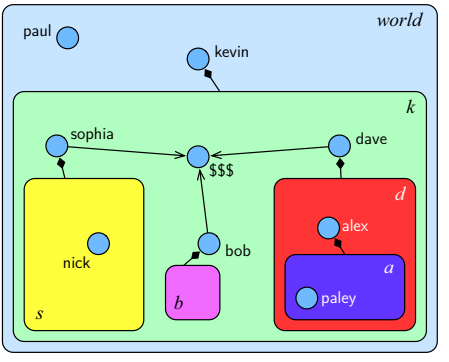


Fig. 3. Sophia and Dave sharing a bank account (**$$$**) in a system with owners-as-dominators. Bob also has access to the account. The legend can be found in Figure 2.

```
class Person<owner> {
  Account<aggregate> account;
  Person<bridge> spouse;
  void share() {
    account = spouse.getAccount();
  }
  Account<aggregate> getAccount() {
    return account;
  }
  ... // omitted
}
```

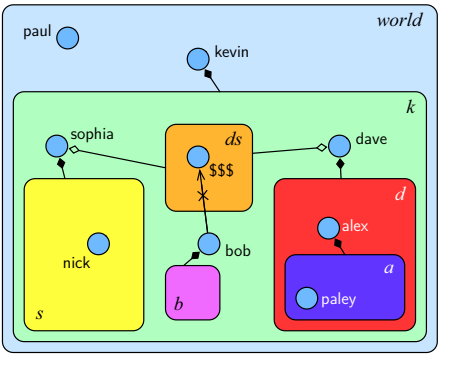


Fig. 4. Sophia and Dave sharing a *private* bank account in a system with owners-as-ombudsmen. The area *ds* denotes a shared context.

Shared Bank Account. Consider a bank account shared between two persons, sophia and dave. Let p be the owner of both sophia and dave, and q be the owner of the bank account. As should be clear by now, in Clarkean systems, the only way in which both sophia and dave can reference the account is if p is nested inside q, meaning that the account has *at most* the same level of encapsulation as sophia or dave. Figure 3 accomplishes this with the resulting ownership graph depicted to the right, which also includes a third person, bob.

For sophia and dave to be able to share an account, there must be a way for sophia to install (*e.g.,* a setter) a reference to the account in dave alternatively a way for dave to read the reference from sophia (*e.g.,* a getter). In either case, any object (*e.g.,* bob) with access to sophia or dave can access the shared account or install another one. Ombudsmen allow us to express the three objects as an aggregate where dave and sophia are in the aggregate's interface whereas the account is private. Figure 4 shows the code and resulting ownership graph.

Schäfer and Poetzsch-Heffter's work on CoBoxes [36,37] include more examples, *e.g.,* a DOM, file system, that need multiple aggregate entry points.

2.2 Ombudsmen

As Figure 4 hints, our proposal allows several objects—*ombudsmen*—to act as bridge objects, or entry points, between their common aggregate and the outside world. In terms of ownership types they share a common context. Objects in this context are "ombudsman-dominated"[1], meaning that

> *every path from a root in the system to an object in an ombudsman-dominated context contains one of the context's ombudsmen.*

As the rightmost picture of Figure 3 shows, in owners-as-dominators systems, every context is nested inside some other context, and references cannot cross a context from the outside to the inside. With *ombudsmen*-as-dominators, objects *on the same level of nesting* can *share a common context*, and references may cross from the "private contexts" of these objects into their shared context.

In the example in Figure 4 (right), sophia and dave are ombudsmen for a collaboratively defined aggregate containing a shared bank account. Their common aggregate context is depicted as the area *ds*. In addition, sophia and dave each have a representation context (*s* and *d*, respectively). The objects sophia and dave, as well as objects in their representation contexts, can reference objects in the aggregate context (*e.g.,* nick may reference $$$). Objects outside sophia or dave may not refer to objects in their aggregate context (*e.g.,* bob may not reference $$$). Objects in the aggregate context cannot reference the representation contexts of their ombudsmen (*e.g.,* $$$ may not refer to alex).

2.3 Typing Ombudsmen

We design the type system that lets us express aggregate encapsulation with multiple entry points as a relatively straightforward extension of Clarkean ownership types systems as found in *e.g.,* Joe$_1$ [13], Joline [14,38] or OGJ [35]. Classes are parametrized over permissions to access external contexts and types instantiate those parameters with actual permissions, so-called *owner parameters*. From now on, we will use the word *owner* to denote a symbol in the program text that represents a run-time context.

The first owner parameter of a type is the owner of the instance, available internally inside each class through the **`owner`** keyword. Additionally, each class knows the owners **`world`**, **`rep`**, **`aggregate`**, and **`bridge`**. The **`world`** keyword denotes the global outermost context. The **`rep`** keyword denotes the representation of the object and is equivalent to **`this`** in traditional ownership systems; **`aggregate`** denotes the aggregate context, which may be shared with other objects; and finally **`bridge`** denotes a bridge object of the same aggregate as the current instance. Notably, if we think of an owner α as denoting the set of objects owned by α, then **`bridge`** $\subseteq$ **`owner`**.

In terms of the rightmost picture in Figure 4, a field in sophia referencing dave (or the inverse) may have the owner **`owner`** or **`bridge`**. A reference from nick to

[1] We slightly abuse the term dominator to stay close to owners-as-dominators.

$$$ must have the owner **aggregate** (from the view of sophia, inside nick it is some other owner parameter which will be bound to sophia's **aggregate**); and sophia's reference to nick must have the owner **rep**.

Whether a field has owner **bridge** or **owner** makes an important difference. In the first case, we know that the field contains a reference to an object sharing the same aggregate. In the second case, we don't know if the reference points to an ombudsman for the same aggregate, or some other aggregate. In terms of our example, sophia could only ask for dave's reference to $$$ if sophia knows dave is a bridge object for *the same* aggregate. Otherwise, the reference might point to the representation of a different aggregate, which would break encapsulation.

Same Object, Different Types. Figure 5 shows an example of a Library component with two provided services with different privileges to access documents, `normalDocAccess` and `privilegedDocAccess`, and a required service `remoteLibrary`. To a client, the objects referred to by `normalDocAccess` and `privilegedDocAccess` are siblings to the component—they have the same owner. From the view of the client, the field `normalDocAccess` has type `AccessService<rep>` rather than `AccessService<bridge>` which would denote a bridge object of the aggregate in the client and not the component. For similar reasons, writing to a field containing an ombudsman is not possible externally, since external objects cannot tell what objects are ombudsmen for the same aggregate.

Discussion. The **bridge** owner identifies other ombudsmen of the *same* aggregate as the aggregate of the current object, which is therefore also an ombudsman.

Well-formed construction of aggregate objects is one of the key considerations of our system design. Any ombudsman has the capability to construct other ombudsmen and access the parts of their interface that mention **aggregate**. All ombudsmen are owned by **owner** (or its more specific subset context **bridge**), and consequently—all dominating objects of the shared aggregate are *siblings*. Coalescing an existing ombudsman object created externally with an aggregate is possible using ownership transfer [14,38,32]. In this case, one object must act as the "initial object" and move the unique objects into **bridge** (cf. Section 2.6).

Although we have defined ombudsmen for the Joe/Joline family of ownership systems [13,14,34,15], we believe they could easily be added to universe types [30,19,32] as well as OGJ [35], and similar.

We now continue our introduction to ombudsmen by way of a few examples including two common programming idioms: components and external iterators.

2.4 Components

Standard UML components are implemented as aggregates of collaborating objects [5]. A component may provide several different interfaces (aka required and provided services); different applications may use different interfaces or a combination of different interfaces.

With ownership types, a component that wishes to export several different interfaces must do so through a single object if encapsulation is to be retained.

```
class Library<owner> {
  DocumentDB<aggregate> db = new DocumentDB<aggregate>;
  AccessService<bridge> normalDocAccess = new RestrictedPolicy<bridge>(db);
  AccessService<bridge> privilegedDocAccess =
                                    new UnrestrictedPolicy<bridge>(db);
  AccessService<owner>  remoteLibrary;
}

class Client<owner> {
  Library<rep> lib, otherLib;
  ...
  AccessService<rep> s1 = lib.normalDocAccess;
  c.remoteLibrary = s1; // = otherLib.normalDocAccess is also type sound
  AccessService<bridge> s2 = lib.normalDocAccess; // Fails!!
  lib.privilegedDocAccess = lib.privilegedDocAccess; // Fails!!
}
```

Fig. 5. Defining a component with two provided services and one required service

If each interface was implemented as a separate object, the objects would not be able to share any data unless that data could also be exposed outside the component, with the problems detailed on Page 160. To implement components with proper encapsulation, several objects must be able to share a common representation which cannot leak, as we did in Figure 4 and Figure 5.

2.5 Iterators with Ombudsmen

Figure 6 shows the ownership diagram for a linked list aggregate and Figure 7 the source code. Modulo the use of the novel **bridge** and **aggregate** owners, the code should be straightforward.

In the example, the list's links are part of the aggregate, and the list has no representation data. Initially, the list object is the only ombudsman through which the links can be manipulated. The `iterator` method in the list class creates and returns an ombudsman in the form of an iterator. As an ombudsman for the list aggregate, the iterator may reference the list's links in the `cursor` field. Any number of iterators may exist; the pattern would even allow several list objects that shared a common set of links—for whatever purpose.

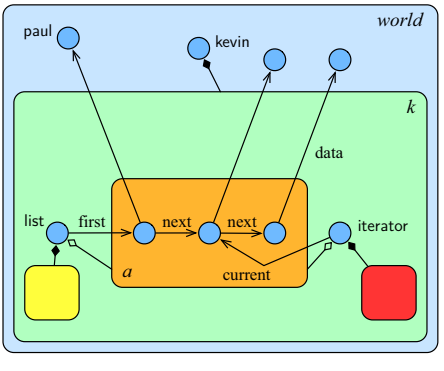


Fig. 6. Iterators with ombudsmen

Notably, in this solution, links are not considered part of the list object's representation, but part of the "list component's" aggregate context, which clearly reflects its degree of encapsulation.

```
class List<owner,data> {
  Link<aggregate,data> first;

  Iterator<bridge,data> iterator() {
    Iterator<bridge,data> iter =
        new Iterator<bridge,data>();
    iter.cursor = first;
    return iter;
  }
}

class Link<owner,data> {
  Link<owner,data> next;
  Object<data> data;
}
```

```
class Iterator<owner,data> {
  Link<aggregate,data> cursor;

  Object<data> next() {
    Object<data> value = cursor.data;
    cursor = cursor.next;
    return value;
  }
}
```

Fig. 7. A list component with a (standard) list service and an iterator service

2.6 Staged Aggregate Construction

Allowing staged construction of aggregates is desirable as it allows *e.g.,* adding user-defined bridges to library aggregates. Attaching an additional bridge object O to an existing aggregate A will merge O's aggregate context, B say, with A.[2] This has consequences for all other bridges to B and unless we provide additional mechanisms for restricting merging of aggregate contexts, we have introduced a back door in the system. Going back to our bank accounts example, if bob can attach himself to dave and sophia's aggregate, then bob can suddenly call the getters and setters for the shared account, thereby gaining access to it.

Allowing the assignment from unique types to bridge types elegantly allows staged construction of an aggregate. Regular non-unique bridge objects cannot be attached to some other aggregate to gain access to its innards. Since an aggregate's **bridge** owner cannot be named externally, the aggregate implementer must explicitly provide a method to perform the attachment. Unless such a method is specified in any of the aggregate's bridges, staged construction is not possible. (This allows preventing bob from using the trick above.)

Aggregates can be constructed incrementally or in stages by attaching ombudsmen to each other, thereby merging their aggregate contexts. This practice is sound as the uniquely referenced ombudsman is always a dominating node to any object inside its aggregate context, akin to the relation between an owner and its **this** context in classical ownership types.

Figure 8 (on Page 165) shows an excerpt of a `List` class that allows an object *external to the aggregate* to be made part of an existing aggregate. In the example, a unique iterator object is passed to the list's iterator method, which subsequently attaches it to its current aggregate by storing it in variable owned by **bridge**.

[2] And unless O is a sibling of A's bridge objects, the attachment is unsound since it would merge differently dominated aggregates.

```
Iterator<bridge,data> iterator(Iterator<unique,data> i) {
  Iterator<bridge,data> iter = i--; // move into bridge
  iter.cursor = first;
  return iter;
}
```

Fig. 8. Attaching an external iterator to a list aggregate; -- is a destructive read

3 Formalizing Ombudsmen

In this section we formalize ombudsmen as an extension to deep ownership. The most relevant changes are in the type rules (EXPR-SELECT), (EXPR-UPDATE) and (EXPR-METHOD-CALL). Our formalism is inspired by [13,38].

When reading or updating a field f of a non-bridge receiver x, f may not point to objects in x's aggregate. This is a straightforward adaptation of the static visibility constraint of Clarkean systems that reads $\mathbf{rep} \in \mathsf{Owners}(\tau) \Rightarrow e = \mathbf{this}$, which our system also uses. Further, if f under the same circumstances points to an ombudsman of x's aggregate, its type is reported to us as a sibling of x. These can be seen in (EXPR-SELECT) and to some extent in (EXPR-UPDATE).

We consider only unary methods for simplicity and without loss of generality. Ombudsman adaptation is employed to translate internal types to external types and there is an additional visibility constraint that prevents calling methods which expect ombudsman arguments, unless the receiver object is itself an ombudsman. The same constraint must hold for field update. This is visible in (EXPR-UPDATE) and (EXPR-METHOD-CALL).

Any type owned by **bridge** can be subsumed by the equivalent type owned by **owner**, since for all contexts, **bridge** denotes a subset of **owner**. This is accomplished by adding an additional subtyping rule, see (BRIDGE-OWNER-SUBSUMPTION).

Conventions and Conveniences. We follow the practice of FJ [25] and use an overbar notation for sequences of terms in the standard fashion. For example, $\overline{p}$ denotes a sequence $p_1, \ldots, p_n$ and $\overline{f} : \overline{\tau}$ denotes a sequence $f_1 : \tau_1, \ldots f_n : \tau_n$ for $n \geq 0$. To turn such a sequence into a set, we write it within $\{\ \}$, *e.g.,* $\{\overline{p}\} = \{p \mid p \in \overline{p}\}$.

Like many ownership types papers before us [16,12,14,38,34], we sometimes write $\mathtt{C}\langle\sigma\rangle$ for a type $\mathtt{C}\langle\overline{p}\rangle$ where σ is a map from the names of the formal parameters of C to the actual owner arguments $\overline{p}$. For example, if C is declared **class** $\mathtt{C}\langle$**owner**, a, b$\rangle$ $\cdots$ in the program, then if $\mathtt{C}\langle p_1, p_2, p_3\rangle$ is a well-formed type, we sometimes write $\mathtt{C}\langle\sigma\rangle$ for the implicitly defined $\sigma = \{\mathbf{owner} \mapsto p_1, \mathtt{a} \mapsto p_2, \mathtt{b} \mapsto p_3\}$. As a further convenience—following previous work—we sometimes write σ^p to mean $\sigma \cup \{\mathbf{owner} \mapsto p\}$ and σ_p to mean $\sigma \cup \{\mathbf{aggregate} \mapsto p\}$ (used in the dynamic semantics, possibly combined with σ^p).

3.1 Static Semantics

Syntax. The syntax of our system is defined in Figure 9. The meta variables x and y are used for names of variables (including **this**) and p and q for names

$$
\begin{array}{lcll}
P & ::= & \overline{C} \quad \texttt{class Object}\langle\texttt{owner}\rangle\ \{\ \}\quad e & (\textit{Program}) \\
C & ::= & \texttt{class C}\langle\texttt{owner}, \overline{p}\rangle\ \texttt{extends D}\langle\overline{p}\rangle\ \{\ \overline{F}\ \overline{M}\ \} & (\textit{Class decl.}) \\
F & ::= & \tau\ f & (\textit{Field decl.}) \\
M & ::= & \tau\ m(\tau\ x)\ \{\ e\ \} & (\textit{Method decl.}) \\
e & ::= & \texttt{let}\ x = e\ \texttt{in}\ e \mid x \mid x.f \mid x.f = y \mid & (\textit{Expressions}) \\
 & & x.m(y) \mid \texttt{null} \mid \texttt{new}\ \tau & \\
\tau & ::= & \texttt{C}\langle\overline{p}\rangle & (\textit{Types})
\end{array}
$$

Fig. 9. Syntax

of contexts (including **rep**, **owner**, **bridge** and **aggregate**). For simplicity, local variables and sequences are encoded using standard let-expressions.

For the static semantics, we use a standard environment E, containing mappings from local variables to types and relations between contexts in the current scope: $E ::= \epsilon \mid E, x : \tau \mid E, p \prec^* q \mid E, p \succ^* q$. Declarations and let-expressions extend E in a straightforward fashion. Table 1 shows an overview of the judgments used in our static system.

Helper Predicates. A key helper predicate is OmbudsmanAdaptation, defined thus:

$$
\begin{array}{rcl}
\mathsf{OmbudsmanAdaptation}(\texttt{bridge}, \tau) & = & \tau \\
\mathsf{OmbudsmanAdaptation}(p, \tau) & = & \tau\{^{\texttt{owner}}/_{\texttt{bridge}}\}
\end{array}
$$

where $p \neq \texttt{bridge}$ is assumed in the last case. This predicate is used to change the internal view of an object as a bridge object of the *current object's* aggregate to the external view of an object—a bridge object for *some* aggregate at the same nesting depth.

For every class C, we can derive a "field table" $\mathcal{FT}(\texttt{C})$ and a "method table" $\mathcal{MT}(\texttt{C})$. We define $\mathcal{FT}(\texttt{C})$ for $\texttt{class C}\langle\overline{p}\rangle\ \texttt{extends D}\langle\sigma\rangle\ \{\ F\ M\ \}$ as $F \bullet \sigma(\mathcal{FT}(\texttt{D}))$ and similar for $\mathcal{MT}(\texttt{C})$. $F(f) = \tau$ if $\tau f \in F$, else $\bot$. Isomorphically, $M(m) = (\tau_1 \to \tau_2, x, e)$ if $\tau_2\ m(\tau_1\ x)\ \{\ e\ \} \in M$, else $\bot$. Lookup in these tables is performed left–right, so $\mathcal{FT}(\texttt{C})(f)$ when $\mathcal{FT}(\texttt{C}) = F \bullet \sigma(\mathcal{FT}(\texttt{D}))$ is defined as $F(f)$ when $f \in \mathrm{dom}(F)$, else $\sigma(\mathcal{FT}(\texttt{D})(f))$. The root class has empty field and method tables; $\mathcal{FT}(\texttt{Object}) = \mathcal{MT}(\texttt{Object}) = \emptyset$ and $\emptyset(f) = \emptyset(m) = \bot$.

Using the tables, we define lookup helper predicates in a straightforward fashion where *first*, *second*, etc. extract the 1st, 2nd, etc. tuple compartments.

$$
\begin{array}{rcl}
\mathsf{FieldType}(\texttt{C}, f) & = & \mathcal{FT}(\texttt{C})(f) \\
\mathsf{Signature}(\texttt{C}, m) & = & \mathcal{MT}(\texttt{C})(m) \\
\mathsf{Param}(\texttt{C}, m) & = & \mathit{second}(\mathcal{MT}(\texttt{C})(m)) \\
\mathsf{Body}(\texttt{C}, m) & = & \mathit{third}(\mathcal{MT}(\texttt{C})(m)) \\
\mathsf{Fields}(\texttt{C}) & = & \{f \mid \mathcal{FT}(\texttt{C})(f) \neq \bot\}
\end{array}
$$

We also define functions for looking up owners from types: $\mathsf{Owners}(\texttt{C}\langle\overline{p}\rangle) = \{\overline{p}\}$ and $\sigma(\texttt{C}\langle\overline{p}\rangle) = \texttt{C}\langle\sigma(\overline{p})\rangle$.

Table 1. Judgments in the static system

$\vdash P : \tau$	P is a well-formed program with type τ
$\vdash C$	C is a well-formed class
$\vdash E, x : \tau$	E is extended by a variable x with type τ
$\vdash E, p \mathrel{\mathcal{R}} q$	E is extended by a good nesting relation ($\mathcal{R} \in \{\prec^*, \succ^*\}$) between contexts p and q
$E; \tau \vdash F$	F is a well-typed field declaration and does not override a field in a supertype
$E; \tau \vdash M$	M is a well-typed method declaration, overriding preserves typing
$E \vdash e : \tau$	e is a well-formed expression with type τ
$E \vdash p$	p is a good owner in the scope E
$E \vdash p \mathrel{\mathcal{R}} q$	p is inside/outside q in the scope E; $\mathcal{R} \in \{\prec^*, \succ^*\}$
$E \vdash p \rightarrow^{\mathsf{ok}} q$	p may reference q in the scope E
$E \vdash \tau$	τ is a well-formed type in the scope E
$E \vdash \tau <: \tau'$	τ is a subtype of τ' in the scope E

Last, we assume the existence of a predicate $\mathsf{Arity}(\mathtt{C})$ that returns the number of owner parameters, including **owner**, declared for the class C, *e.g.,* $\mathsf{Arity}(\mathtt{List}) = 2$ from the example in Figure 7.

Declarations. A program is well-formed if all its classes are well-formed and the starting expression of the program is well-typed. For simplicity, the root class Object is treated special.

$$\text{(WF-PROGRAM)}\quad \frac{\vdash \overline{C} \qquad \epsilon \vdash e : \tau}{\vdash \overline{C} \quad \mathbf{class}\ \mathtt{Object}\langle \mathbf{owner} \rangle\ \{\ \}\quad e : \tau}$$

A class is well-formed if its fields and methods are well-formed, the owner parameters passed to the super class are good (respect the nesting), and **owner** is only used in the first position of the owner formals.

$$\text{(WF-CLASS)}\quad \frac{\begin{array}{c} E = \mathbf{owner} \prec^* \mathbf{world}, \mathbf{rep} \prec^* \mathbf{owner}, \mathbf{bridge} \prec^* \mathbf{owner}, \backslash \\ \mathbf{aggregate} \prec^* \mathbf{owner}, \overline{p} \succ^* \mathbf{owner}, \mathbf{this} : \mathtt{C}\langle \mathbf{bridge}, \overline{p} \rangle \qquad \{\overline{q}\} \subseteq \{\overline{p}\} \\ \mathbf{owner} \notin \{\overline{p}\} \qquad \tau_{\mathsf{s}} = \mathtt{D}\langle \mathbf{owner}, \overline{q} \rangle \qquad E \vdash \tau_{\mathsf{s}} \qquad E; \tau_{\mathsf{s}} \vdash \overline{F} \qquad E; \tau_{\mathsf{s}} \vdash \overline{M} \end{array}}{\vdash \mathbf{class}\ \mathtt{C}\langle \mathbf{owner}, \overline{p} \rangle\ \mathbf{extends}\ \mathtt{D}\langle \overline{q} \rangle\ \{\ \overline{F}\ \overline{M}\ \}}$$

A field is well-typed if its type is valid in the current scope, and there is no field with the same name in a superclass.

$$\text{(WF-FIELD)}\quad \frac{E \vdash \mathtt{C}\langle \sigma \rangle \qquad E \vdash \tau \qquad \mathsf{FieldType}(\mathtt{C}, f) = \bot}{E; \mathtt{C}\langle \sigma \rangle \vdash \tau\ f}$$

A method is well-formed if its types are well-formed in the current scope, its body corresponds to the declared return type, and overriding preserves types.

$$\frac{\begin{array}{c}E \vdash \mathtt{C}\langle\sigma\rangle \qquad E \vdash \tau \qquad E, x : \tau' \vdash e : \tau \\ \mathsf{Signature}(\mathtt{C}, m) = \bot \vee \mathsf{Signature}(\mathtt{C}, m) = \sigma(\tau') \to \sigma(\tau)\end{array}}{E; \mathtt{C}\langle\sigma\rangle \vdash \tau\ m(\tau'\ x)\ \{\ e\ \}}\ (\textsc{wf-method})$$

Expressions. Expressions are typed given the type information E derived initially for each method in (WF-CLASS), and extended with variables by (WF-METHOD) and (EXPR-LET). For simplicity, we assume that all variables have unique names.

$$\frac{E \vdash e' : \tau' \quad E, x : \tau' \vdash e : \tau}{E \vdash \mathtt{let}\ x = e'\ \mathtt{in}\ e : \tau}\ (\textsc{expr-let}) \qquad \frac{\vdash E \qquad E(x) = \tau}{E \vdash x : \tau}\ (\textsc{expr-var})$$

Reading a field of an object makes use of two key constraints: first, the two visibility constraints that say representation objects may only be accessed through the special `this` receiver, which is due to Clarke et al. [16], and that aggregate objects may only be accessed through ombudsmen. Last, we apply the OmbudsmanAdaptation helper function that make ombudsmen appear as regular objects when viewed externally.

$$\frac{\begin{array}{c}E \vdash x : \mathtt{C}\langle\sigma^p\rangle \qquad \mathsf{FieldType}(\mathtt{C}, f) = \tau \\ \mathtt{rep} \in \mathsf{Owners}(\tau) \Rightarrow x = \mathtt{this} \\ \mathtt{aggregate} \in \mathsf{Owners}(\tau) \Rightarrow p = \mathtt{bridge} \\ \mathsf{OmbudsmanAdaptation}(p, \tau) = \tau'\end{array}}{E \vdash x.f : \sigma^p(\tau')}\ (\textsc{expr-select})$$

(EXPR-UPDATE) is very similar to (EXPR-SELECT), although it does not use OmbudsmanAdaptation (that would not be sound as we are writing, not reading a field—similar to wild-cards in Java generics) and adds an additional restriction: a field holding an ombudsman can only be accessed through another ombudsman.

$$\frac{\begin{array}{c}E \vdash x : \mathtt{C}\langle\sigma^p\rangle \qquad \mathsf{FieldType}(\mathtt{C}, f) = \tau \qquad E \vdash y : \sigma^p(\tau) \\ \mathtt{rep} \in \mathsf{Owners}(\tau) \Rightarrow x = \mathtt{this} \\ \mathtt{bridge}, \mathtt{aggregate} \in \mathsf{Owners}(\tau) \Rightarrow p = \mathtt{bridge}\end{array}}{E \vdash x.f = y : \sigma^p(\tau)}\ (\textsc{expr-update})$$

The semantics for calling a method is straightforward and contains the amalgamation of the restrictions of (EXPR-SELECT) and (EXPR-UPDATE) as well as uses OmbudsmanAdaptation so that returning a `bridge` object from an invocation on a non-bridge object type loses its "bridge status" (in the type system's view).

$$\frac{\begin{array}{c}(\textsc{expr-method-call})\\ E \vdash x : \mathtt{C}\langle\sigma^p\rangle \qquad \mathsf{Signature}(\mathtt{C}, m) = \tau_1 \to \tau_2 \qquad E \vdash y : \sigma^p(\tau_1)\\ \mathtt{rep} \in \mathsf{Owners}(\tau_1) \cup \mathsf{Owners}(\tau_2) \Rightarrow x = \mathtt{this}\\ \mathtt{bridge}, \mathtt{aggregate} \in \mathsf{Owners}(\tau_1) \Rightarrow p = \mathtt{bridge}\\ \mathtt{aggregate} \in \mathsf{Owners}(\tau_2) \Rightarrow p = \mathtt{bridge}\\ \mathsf{OmbudsmanAdaptation}(p, \tau_2) = \tau\end{array}}{E \vdash x.m(y) : \sigma^p(\tau)}$$

The static semantics for `null` and instantiation are straightforward. Last, (expr-subsumption) allows the type of an expression to be subsumed into a supertype.

$$\frac{E \vdash \tau}{E \vdash \mathtt{null} : \tau}\ (\textsc{expr-null}) \qquad \frac{E \vdash \tau}{E \vdash \mathtt{new}\ \tau : \tau}\ (\textsc{expr-new}) \qquad \frac{E \vdash e : \tau' \qquad E \vdash \tau' <: \tau}{E \vdash e : \tau}\ (\textsc{expr-subsumption})$$

Type Environment Construction. We use a standard static type environment E.

$$\frac{}{\vdash \epsilon}\ (\textsc{e-}\epsilon) \qquad \frac{E \vdash \tau \qquad x \notin dom(E)}{\vdash E, x : \tau}\ (\textsc{e-var}) \qquad \frac{E \vdash q \qquad p \notin dom(E) \qquad \mathcal{R} \in \{\prec^*, \succ^*\}}{\vdash E, p\ \mathcal{R}\ q}\ (\textsc{e-context})$$

Contexts. Statically, contexts are added to the environment in (wf-class). The only manifest owner is `world`.

$$\frac{\vdash E \qquad p \in dom(E)}{E \vdash p}\ (\textsc{good-context}) \qquad \frac{\vdash E}{E \vdash \mathtt{world}}\ (\textsc{good-world})$$

Rules for nesting relations are straightforward and follow a wealth of ownership papers in the Clarkean family.

$$\frac{\vdash E \qquad p \prec^* q \in E}{E \vdash p \prec^* q}\ (\textsc{inside}) \qquad \frac{\vdash E \qquad q \succ^* p \in E}{E \vdash p \prec^* q}\ (\textsc{outside}) \qquad \frac{E \vdash p}{E \vdash p \prec^* p}\ (\textsc{inside-reflexive}) \qquad \frac{E \vdash p \prec^* p' \qquad E \vdash p' \prec^* q}{E \vdash p \prec^* q}\ (\textsc{inside-transitive})$$

Permissions. Permissions in our system govern how references may cross context boundaries. Inside nesting implies permission to reference, just like in classical ownership types in (p-inside).

$$\frac{E \vdash p \prec^* q}{E \vdash p \to^{\mathsf{ok}} q}\ (\textsc{p-inside}) \qquad \frac{\vdash E \qquad p \in \{\mathtt{bridge}, \mathtt{aggregate}\}}{E \vdash \mathtt{rep} \to^{\mathsf{ok}} p}\ (\textsc{p-rep})$$

An ombudsman's representation may reference its aggregate in (p-rep).

Types and Subtyping. In our system, a type is well-formed if its owner has the right to reference all its owner parameters, and additionally, the number of parameters must correspond to the class declaration.

$$\frac{E \vdash p \qquad E \vdash p \rightarrow^{\mathsf{ok}} \overline{p} \qquad \mathsf{Arity}(\mathtt{C}) = |p, \overline{p}|}{E \vdash \mathtt{C}\langle p, \overline{p}\rangle} \text{(GOOD-TYPE)}$$

Subtyping follows the same rules as for classic ownership types. Reference permissions are propagated upward in the class hierarchy by the forwarding in the class declaration, and the subtyping relation is reflexive and transitive.

$$\frac{E \vdash \mathtt{C}\langle\sigma\rangle \qquad \mathtt{class}\ \mathtt{C}\langle\cdots\rangle\ \mathtt{extends}\ \mathtt{D}\langle\overline{q}\rangle \cdots \in P}{E \vdash \mathtt{C}\langle\sigma^p\rangle <: \mathtt{D}\langle p, \sigma(\overline{q})\rangle} \text{(SUBTYPE-DIRECT)}$$

$$\frac{E \vdash \tau}{E \vdash \tau <: \tau} \text{(SUBTYPE-REFLEXIVE)}$$

$$\frac{E \vdash \tau_1 <: \tau_3 \qquad E \vdash \tau_3 <: \tau_2}{E \vdash \tau_1 <: \tau_2} \text{(SUBTYPE-TRANS)}$$

The single novel subtyping rule in our system allows an ombudsman to be subsumed by its owner. This is required to safely export an ombudsman outside of its (aggregate) representation without compromising safety.

$$\frac{E \vdash \mathtt{C}\langle\mathtt{bridge}, \overline{p}\rangle \qquad E \vdash \mathtt{C}\langle\mathtt{owner}, \overline{p}\rangle}{E \vdash \mathtt{C}\langle\mathtt{bridge}, \overline{p}\rangle <: \mathtt{C}\langle\mathtt{owner}, \overline{p}\rangle} \text{(BRIDGE-OWNER-SUBSUMPTION)}$$

As an example of the use of this practice, see the list iterator example. Internally, the list's view of its iterator is `Iterator⟨bridge, data⟩`, but when obtained from some external object, the iterator's type is `Iterator⟨owner, data⟩`. This is sound since **bridge** always denotes a subset of **owner**.

3.2 Dynamic Semantics

The dynamic semantics is a big-step operational semantics. To distinguish diverging computation from stuck states, we use a standard trick to limit stack space [21,38]. Each reduction carries the remaining stack space and each method call reduces this number. A method call when there is no remaining stack space triggers an error.

Objects are represented by triples of type, aggregate context id α, and field mappings. Run-time types are the same as static types, but static owner names are substituted for run-time contexts. Run-time contexts are κ (an object id ι, aggregate context id α, or the special context **world**). Values are ι or ϵ (null).

$$\begin{array}{lcll} H & ::= & [\,] \mid H[\iota \mapsto (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, F)] & (\textit{Heap}) \\ B & ::= & \epsilon \mid B, x \mapsto v \mid B, p \mapsto \kappa & (\textit{Bindings}) \\ \mathcal{F} & ::= & [\,] \mid \mathcal{F}[f \mapsto v] & (\textit{Fields}) \end{array} \qquad \begin{array}{lcll} v & ::= & \epsilon \mid \iota & (\textit{Values}) \\ \kappa & ::= & \iota \mid \alpha \mid \mathtt{world} & (\textit{Contexts}) \end{array}$$

A configuration is a triple $\langle H; B; e\rangle$ of a heap H, bindings of variables to values and context names to contexts B, and an expression e. The initial configuration is $\langle[\,]; \emptyset; e\rangle$ that is, an empty heap and bindings, plus the initial expression.

Rules (D-LET) and (D-VAR) are unsurprising. (D-LET) evaluates the expression e and binds the value v' to the variable x in the environment under which e' is evaluated. (D-VAR) just looks up the value bound to x in the frame.

$$\frac{\langle H; B; e\rangle \ \rightarrow_n \ \langle H'; v'\rangle \quad \langle H'; B, x \mapsto v'; e'\rangle \ \rightarrow_n \ \langle H''; v''\rangle}{\langle H; B; \texttt{let}\ x = e\ \texttt{in}\ e'\rangle \ \rightarrow_n \ \langle H''; v''\rangle}\ (\text{D-LET}) \qquad \frac{B(x) = v}{\langle H; B; x\rangle \ \rightarrow_n \ \langle H; v\rangle}\ (\text{D-VAR})$$

Looking up a field on an object receiver is straightforward. We write $H(\iota.f)$ as a shorthand for $\mathcal{F}(f)$ when $H(\iota) = (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F})$. Field updates are similar, and we write $H(\iota.f) := v$ for $H[\iota \mapsto (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F}[f \mapsto v])]$ when $H(\iota) = (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F})$.

$$\frac{B(x) = \iota \qquad H(\iota.f) = v}{\langle H; B; x.f\rangle \ \rightarrow_n \ \langle H; v\rangle}\ (\text{D-SELECT}) \qquad \frac{B(x) = \iota \qquad B(y) = v}{\langle H; B; x.f = y\rangle \ \rightarrow_n \ \langle H(\iota.f) := v; \epsilon\rangle}\ (\text{D-UPDATE})$$

In our simple semantics, method calls cause the evaluation of a method body under a new B with all owner names substituted for their run-time equivalents, derived from the current `this`. Furthermore, `this` is substituted for the current object, and the parameter is substituted for the actual argument value.

$$\frac{\begin{array}{c} B(x) = \iota \qquad B(y) = v \qquad H(\iota) = (\mathtt{C}\langle\sigma^{\kappa}\rangle, \alpha, \_) \\ \mathsf{Body}(\mathtt{C}, m) = e \qquad \mathsf{Param}(\mathtt{C}, m) = x \\ B' = \texttt{rep} \mapsto \iota, \texttt{bridge} \mapsto \kappa, \texttt{this} \mapsto \iota, x \mapsto v, \texttt{aggregate} \mapsto \alpha \\ n > 0 \qquad \langle H; B', \sigma^{\kappa}; e\rangle \ \rightarrow_{(n-1)} \ \langle H'; v'\rangle \end{array}}{\langle H; B; x.m(y)\rangle \ \rightarrow_n \ \langle H'; v'\rangle}\ (\text{D-METHOD-CALL})$$

The run-time representation of `null` is denoted by ϵ. Object creation is simple due to the absence of constructors and custom field initialization. A fresh object has all its fields initialized to `null` and a fresh context α is picked to represent its aggregate, unless it is a ombudsman, in which case the aggregate context is that of the current object.

$$\frac{}{\langle H; B; \texttt{null}\rangle \rightarrow_n \langle H; \epsilon\rangle}\ (\text{D-NULL}) \qquad \frac{\begin{array}{c} \mathcal{F} = [f \mapsto \epsilon \mid f \in \mathsf{Fields}(\mathtt{C})] \qquad \iota \text{ is fresh} \\ p \neq \texttt{bridge} \Rightarrow \alpha \text{ fresh} \quad p = \texttt{bridge} \Rightarrow \alpha = B(\texttt{aggregate}) \end{array}}{\langle H; B; \texttt{new}\ \mathtt{C}\langle p, \overline{p}\rangle\rangle \ \rightarrow_n \ \langle H[\iota \mapsto (\mathtt{C}\langle B(p), B(\overline{p})\rangle, \alpha, \mathcal{F})]; \iota\rangle}\ (\text{D-NEW})$$

For brevity, we omit the trivial error trapping rules for dereferencing null pointers and propagating errors and stack overflow.

3.3 Meta Theory

In our reasoning about well-formedness, we rely on a combined type environment and store type $\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, \iota : \tau \mid \Gamma, \alpha : \kappa \mid \Gamma, \mathtt{o}\,\iota : \alpha$. The entry $\alpha : \kappa$

Table 2. Judgments in the meta-theoretic part of the formalism

$\vdash \Gamma$	Γ is a well-formed store type
$\Gamma \vdash \langle H; B; e\rangle : \tau$ $\Gamma \vdash \langle H; B; v\rangle : \tau$	$\langle H; B; e/v\rangle$ is a well-formed configuration with type τ under Γ
$\Gamma \vdash \mathtt{C}\langle\overline{\kappa}\rangle$	$\mathtt{C}\langle\overline{\kappa}\rangle$ is a well-formed type under Γ
$\Gamma \vdash \kappa \rightarrow^{\mathsf{ok}} \kappa'$	Objects in context κ have permission to reference objects immediately in κ' under Γ
$\Gamma \vdash H$	H is a well-formed heap under Γ
$\Gamma \vdash v : \tau$	Value v has type τ under Γ

maps an aggregate context α to the owner κ of all its ombudsmen. In a similar fashion, the entry $\mathtt{o}\,\iota : \alpha$ maps an object ι into an aggregate context α for which it acts as an ombudsman. Table 2 overviews the judgments in the meta theory.

$$\frac{}{\vdash \epsilon}\ (\Gamma\text{-}\epsilon) \qquad \frac{x \notin dom(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}\ (\Gamma\text{-VAR}) \qquad \frac{\iota \notin dom(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, \iota : \tau}\ (\Gamma\text{-OBJECT})$$

The rules (Γ-BRIDGE) and (Γ-AGGREGATE) are key elements here; in a well-formed store type, all ombudsmen of the same aggregate have the same owner.

$$\frac{\Gamma \vdash \kappa \qquad \alpha \notin dom(\Gamma)}{\vdash \Gamma, \alpha : \kappa}\ (\Gamma\text{-BRIDGE}) \qquad \frac{\vdash \Gamma \qquad \mathtt{o}\,\iota \notin dom(\Gamma) \qquad \alpha : \kappa \in \Gamma \qquad \Gamma(\iota) = \mathtt{C}\langle\sigma^{\kappa}\rangle}{\vdash \Gamma, \mathtt{o}\,\iota : \alpha}\ (\Gamma\text{-AGGREGATE})$$

A well-formed heap can be extended by an object whose field contents correspond to that of the class declaration. All ombudsmen for the same aggregate must have the same owner.

$$\frac{\vdash \Gamma}{\Gamma \vdash [\,]}\ (\text{HEAP-}[]) \qquad \frac{\begin{array}{c}\Gamma(\iota) = \mathtt{C}\langle\sigma^{\kappa}\rangle \qquad \Gamma(\mathtt{o}\,\iota) = \alpha \qquad \Gamma(\alpha) = \kappa \qquad \Gamma \vdash H \\ \Gamma \vdash \overline{v} : (\sigma^{\kappa}_{\alpha} \cup \{\mathtt{rep} \mapsto \iota, \mathtt{bridge} \mapsto \kappa\})(\overline{\tau}) \\ \mathsf{Fields}(\mathtt{C}) = \{\overline{f}\} \qquad \mathsf{FieldType}(\mathtt{C}, \overline{f}) = \overline{\tau}\end{array}}{\Gamma \vdash H, \iota \mapsto (\mathtt{C}\langle\sigma^{\kappa}\rangle, \alpha, [\overline{f} \mapsto \overline{v}])}\ (\text{HEAP-OBJECT})$$

A configuration is well-formed given an environment Γ if its heap is well-formed and its expression/value is well-typed.

$$\frac{\Gamma \vdash H \qquad \Gamma \vdash e\,\{B\} : \tau\,\{B\}}{\Gamma \vdash \langle H; B; e\rangle : \tau}\ (\text{GOOD-CONFIGURATION}) \qquad \frac{\Gamma \vdash H \qquad \Gamma \vdash v : \tau\,\{B\}}{\Gamma \vdash \langle H; B; v\rangle : \tau}\ (\text{GOOD-FINAL-CONFIGURATION})$$

We assume a function $e/\tau\,\{B\}$ that replaces static names of owners in the domain of B with their dynamic counterparts, *e.g.,* $\mathtt{C}\langle p\rangle\,\{B\} = \mathtt{C}\langle B(p)\rangle$. The judgments

$\Gamma \vdash e : \tau$ are "copy-and-patch" from the corresponding type rules $E \vdash e : \tau$ and therefore omitted.

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \epsilon : \tau}\ (\textsc{null-type}) \qquad \frac{\Gamma(\iota) = \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \iota : \tau}\ (\textsc{object-type})$$

Rules for good dynamic contexts are similar to their static counterparts. A type is well-formed if its owner has the right to reference all other owner parameters.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{world}}\ (\textsc{context-world}) \qquad \frac{\vdash \Gamma \qquad \kappa \in dom(\Gamma)}{\Gamma \vdash \kappa}\ (\textsc{context-object/aggregate}) \qquad \frac{\Gamma \vdash \kappa \rightarrow^{\mathsf{ok}} \overline{\kappa} \qquad \mathsf{Arity}(\mathtt{C}) = |\kappa, \overline{\kappa}|}{\Gamma \vdash \mathtt{C}\langle \kappa, \overline{\kappa} \rangle}\ (\textsc{d-type})$$

Except for aggregates and bridges, relations between contexts are not explicitly stored in Γ. Instead we infer them from the types present in a well-formed Γ. Reflexivity and transitivity of this relation are trivial and therefore omitted.

$$\frac{\vdash \Gamma \qquad \Gamma(\kappa) = \mathtt{C}\langle \overline{\kappa} \rangle \qquad \kappa' \in \overline{\kappa}}{\Gamma \vdash \kappa \prec^* \kappa'}\ (\textsc{d-inside}) \qquad \frac{\vdash \Gamma \qquad \Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha \prec^* \kappa}\ (\textsc{d-ombudsman})$$

Last, a context κ may reference another context κ' if an inside relation can be inferred from the first to the second, or if the second is an aggregate context and the first is inside an object defining it.

$$\frac{\Gamma \vdash \kappa \prec^* \kappa' \quad \vee \quad \Gamma \vdash \kappa \prec^* \iota \wedge \Gamma(\mathtt{o}\,\iota) = \kappa'}{\Gamma \vdash \kappa \rightarrow^{\mathsf{ok}} \kappa'}\ (\textsc{d-mayref})$$

We define $\rightarrow$ ("points to") and $\sqsupseteq$ ("aggregates") as binary relations between objects for some heap H such that $\iota \rightarrow \iota' \iff \exists\, f \; s.t. \; H(\iota.f) = \iota'$ and $\iota \sqsupseteq \iota' \iff H(\iota) = (\mathtt{C}\langle \overline{\kappa} \rangle, \alpha, \_) \wedge H(\iota') = (\mathtt{C}\langle \alpha, \_ \rangle, \_, \_)$. We can now define "ombudsmen-as-dominators" as a straightforward extension to owners-as-dominators.

Theorem 1: Ombudsmen-as-dominators. For any two objects ι_1, ι_2 in a well-formed heap, $\iota_1 \rightarrow \iota_2 \Rightarrow \iota_1 \prec^* \mathsf{Owner}(\iota_2) \vee \exists\, \iota$ s.t. $\iota \sqsupseteq \iota_2 \wedge \iota_1 \prec^* \iota$.

> In plain English this states that if an object ι_1 references another object ι_2, then either the owner of ι_2 is a dominator of ι_1 (this is the standard owners-as-dominators property, the non-savvy ownership reader can consult *e.g.,* Clarke's dissertation [12] for additional details), or ι_2 is part of some aggregate and ι_1 is inside the representation of this aggregate.

Proof. Assume $\Gamma \vdash H$ in which $\iota_1 \rightarrow \iota_2$. Let the run-time type of ι_1 be some type $\mathtt{C}\langle \overline{k} \rangle$. By (HEAP-OBJECT), ι_2 is owned by some owner, k say, in $\overline{k}$, or the run-time values for **world**, **rep** (ι_1), **bridge** (ι_1's owner) and **aggregate** (α, say).

First, let k' be the owner of ι_1. By (D-INSIDE) follows $\iota_1 \prec^* k'$ and by (D-TYPE) follows $k' \rightarrow^{\mathsf{ok}} k$ for any k in $\overline{k}$. Now, by (D-MAYREF), either $k' \prec^* k$ (implying $\iota_1 \prec^* k$ by transitivity), or exists ι s.t. $\iota_1 \prec^* \iota$ (by transitivity) and $\Gamma(\mathtt{o}\,\iota) = k'$, which implies $\iota \sqsupseteq \iota_2$. (∗) If $k = \mathtt{world}$ then $\iota_1 \prec^* k$ by definition. If $k = \iota_1$, then $\iota_1 \prec^* k$ by reflexivity of $\prec^*$. The case when $k = i_1$'s owner is subsumed by the above since ι_1's owner is in $\overline{k}$. If $k = \alpha$, then second part of the theorem's implication holds for $\iota = \iota_1$ (∗∗)

For clarity, in (∗), ι_1 belongs to the representation of an object that aggregates ι_2. In (∗∗), ι_1 aggregates ι_2.

Theorem 2: Subject Reduction We prove subject reduction in the standard fashion of progress plus preservation.

PROGRESS: If $\Gamma \vdash \langle H; B; e\rangle : \tau$, then $\langle H; B; e\rangle \rightarrow_n \langle H'; v\rangle$ or $\langle H; B; e\rangle \rightarrow_n \mathsf{ERR}$ for some finite value of n.

Proof. The proof is straightforward by induction on the big-step rules where most cases are immediate. The slightly more intricate cases, (D-SELECT), (D-UPDATE) and (D-METHOD-CALL) are all guarded by versions of the same rule (elided in this presentation) that capture null-dereferencing or stack overflow. By (HEAP-OBJECT), a well-formed object $(\tau, \alpha, \mathcal{F})$ has all its expected fields in $\mathcal{F}$, with the expected types, therefore, evaluation cannot get stuck accessing a non-existent field, and a similar argument applies to method calls. □

PRESERVATION: If $\Gamma \vdash \langle H; B; e\rangle : \tau$ and $\langle H; B; e\rangle \rightarrow \langle H'; v\rangle$, then there exists $\Gamma' \supseteq \Gamma$ s.t. $\Gamma' \vdash \langle H'; B; v\rangle : \tau$ (omitting stack space for simplicity).

Proof. The proof is straightforward by structural induction on the shape of e. There are no surprising cases. (Although B might be updated by evaluating e, such updates will only be of local variables—not owners, which are the interesting elements of B w.r.t. final configurations.) □

4 Related Work

Several researchers have proposed relaxations of Clarkean ownership types that can be used to overcome the single bridge object-problem. The main difference with these systems is that ombudsmen explicate the notions of aggregates and bridges in the types, *e.g.,* allowing an object to clearly identify other bridges to its aggregate, and work within the confines of owners-as-dominators, rather than providing a "back door" which allow external access to an object's internals.

Boyapati et al. [7] allow relaxing owners-as-dominators for instances of inner classes. A list may define an inner iterator class that can be exported arbitrarily, but still access the enclosing object's representation. This allows expressing mutating and non-mutating iterators, but at the same time destroys the strong encapsulation, as there is no way for a type system to detect whether a back door to an object's representation exists or not.

Boyapati's proposal is somewhat close in spirit to ours: a single object starts as the initial bridge object for an aggregate, and may create additional bridge objects internally. However, a closer look reveals several shortcomings, which our system avoids:

Non-modular. All bridge objects must be defined within a single lexical scope. This destroys separate compilation, and also prevents reusing external classes for bridge objects (*e.g.,* it is not possible to have a common iterator class for different list classes).

Inflexible. The initial bridge object must always be the outermost enclosing class of the classes defining an aggregate. This is inflexible as it does not allow defining an aggregate which can be created in multiple ways, using different classes (*e.g.,* a component with different ports for different configurations). Also, staged construction of aggregates is not possible.

State confusion. There is no support for distinguishing between the representation of the initial bridge object (private implementation details) and the aggregate's representation (which might be exported to another bridge). Consequently, an iterator can leak details about the list which were not intended to be exported. Strangely enough, the iterator can have representation which is private from the list.

Ad hoc encapsulation. Boyapati's bridge objects can be exported arbitrarily high up in the nesting hierarchy, making it hard to reason about the origins of changes and completely destroying strong encapsulation.

The strength of Boyapati's proposal is the ability to allow bridge objects to escape arbitrarily outside their defining aggregate. The downside of this flexibility is lack of flexibility in all other domains and the unclear guarantees that this built-in back door gives the system.

OIGJ [40] take Boyapati's approach and suffers from all but the last problem above. In OIGJ, inner classes have access to the representation of the enclosing object. The difference to Boyapati's system is that in OIGJ the inner class must have the same owner as the outer object (*e.g.,* iterator has same owner as its list), and thus cannot be arbitrarily exported, similar to our system. There is however no way to distinguish bridges of a common aggregate from other peers. As in Boyapati's system the outer object grants subsequent bridge objects access to all its state, there is no distinguishing of aggregate and bridge object representation.

The encapsulation property of Universe Types [30], owners-as-modifiers, relaxes owners-as-dominators for read-only references. Thus, traditional universe types can express the iterator pattern, but only allow obtaining read-only references to the list elements via an iterator (modulo expensive and unsafe downcasts). Generic Universe Types [19] overcome this limitation, but do not allow iterators that change its originating collection. In summary, Universe types allow multiple entry points to aggregates, but only one of these entry points may have mutating capabilities. Furthermore, the multiple entry points can be exported arbitrarily in the system. In a concurrent setting this may not be desirable as read-only references do not preclude the existence of mutable aliases, making

them subject to possible data races. Clarke et al. [15] similarly relax owner-as-dominators but for *safe* references (that may only be used to read immutable parts of objects), and for references to immutable data, guaranteeing that no mutation may occur concurrently.

Lu et al. [29] overcome some of the limitations of Boyapati's escaping inner classes by allowing dynamically exposing internal representation through a "downgrading" operation. This voids the need of a specific inner class for exposure, which allows separate compilation and reuse, but just like with inner classes, their downgrading operation destroys the strong notion of encapsulation resulting in unclear properties of the resulting system. Shallow ownership (*e.g.,* [2]) is reminiscent of downgrading in that an object internal to an aggregate can arbitrarily pass on permission to reference aggregate objects to an external object that it creates. Shallow ownership however has no strong (or clear) encapsulation guarantee.

Ownership Domains [1] allows a programmer to manually specify contexts and how objects in these contexts may refer to each other by linking them together. For instance, a list class may define a public domain for its iterators, which is linked to both the domain containing the list links and the element domain. While this is straightforward and flexible, it is difficult to identify the encapsulation invariants of a system: it is necessary to look at large parts of a system and how its components introduce new links between contexts that would invalidate assumptions about encapsulation drawn locally by just studying the list class. Ownership domains further suffer from problems similar to Boyapati's inner classes in that public domains are publicly accessible, and therefore an iterator may be arbitrarily exported.

CoBoxes [36] (and JCoBoxes [37]) are active-objects-like systems with asynchronous message sends and futures. CoBoxes are similar to our aggregates in that they are defined in terms of the objects they contain and may have multiple entry points into an aggregate. However, both CoBoxes and JCoBoxes rely entirely on run-time checks to protect a box's innards, whereas our system can express and check fortified aggregates with multiple entry points at compile-time.

MOJO [11] and Mojojojo [28][3] support multiple ownership, which is more general than our proposal, but this flexibility comes with very high complexity. The descriptive nature of MOJO and Mojojojo allows a programmer to express an aggregate in the types, but encapsulation is not enforced. Further, because the aggregate is visible (in the types) from the outside it can be constructed, populated and extended from the outside. In our system such operations are under complete control of the aggregate itself.

In the context of verification of object invariants, Barnett and Naumann [4] define a friendship protocol in which a granting class can give privileges to another friend class that allows invariants in the friend to depend on fields in the granting class. Objects are connected using an explicit attach construct, but there is no notion of collaboratively defined state, and once a value of a field

[3] Although MOJO and Mojojojo differ in expressiveness and technical details they are very similar in spirit, so we treat them as one here.

in a granting class has been obtained by a friend, the value may be exported arbitrarily.

Boyland et al. [10] use effects and a novel "from" annotation to allow a data structure to temporarily yield its state to an external object (*e.g.,* a list to an iterator). Such access permissions are treated linearly, and therefore cannot be used to express multiple (read/write) entry points to an aggregate, including fail-fast iterators with mutating capabilities and our shared bank account.

Lastly, Joe$_1$ by Clarke and Drossopoulou [13] allow final variables to be used as owners to externally name an object's representation. This relaxation is however only made for variables on the stack and therefore cannot be used to express multiple entry points to aggregates in a straightforward fashion.

5 Discussion

Ombudsmen-as-dominators is a straightforward extension to owners-as-dominators: the owners-as-dominators property holds for all objects in the **rep** context; objects inside **aggregate** contexts are instead dominated by an unknown subset of objects of the directly enclosing context. Thus, objects inside an aggregate context enjoy a weaker encapsulation than representation objects which is precisely the intention of our proposal since many aggregates cannot be expressed in the hierarchical fashion that deep ownership types dictate. This has consequences for reuse and computational effects, which is discussed below.

5.1 Ombudsmen and Reuse

Internally, an object will not know whether it lives inside another object's representation, or constructs an aggregate, which allows programmers to design objects without concern for how they will be used in future systems. Consequently, an object cannot know whether it is dominated by *a single* object or a *collection of several* objects (which would presumably violate abstraction), but we have not yet seen a programming pattern where this is an important factor.

A drawback of our system is that a class cannot be retrofitted to be an ombudsman unless it makes use of the **aggregate** context. Removing this restriction is simple, just give bridge objects implicit permission to reference aggregate objects, and involves the addition of a single type rule:

$$\frac{\vdash E}{E \vdash \texttt{bridge} \rightarrow^{\mathsf{ok}} \texttt{aggregate}}\ (\textsc{p-ombudsman})$$

This causes a problem with presenting a type of an ombudsman external to the aggregate, since there is no external name for the aggregate context. This can be solved using "lost owners" [18]. The type C⟨**bridge**, **aggregate**⟩ will externally be C⟨**owner**, ?⟩ where ? is an owner that cannot be named in the current context.

5.2 Ombudsmen and Ownership-Based Computational Effects

The idea of ombudsmen was conceived during our work on extending Joëlle [15], a language for safe, reliable and efficient parallel programming based on the active object pattern [23]. To achieve the necessary isolation for active objects, Joëlle relies on an "flat" ownership types system where ownership forms a forest and every active object is a root of a tree in the forest.

Our extension to Joëlle sports a type and effect system which is an amalgamation of Greenhouse and Boyland's OOFX [24] and Clarke and Drossopoulou's Joe_1 together with support for externally unique (from Wrigstad's Joline [14,38]), immutable and safe [33] references.

In Section 2.6, we showed how external uniqueness can be used to allow the external creation of a bridge object for an aggregate without introducing back doors. Owner-polymorphic methods, such as found in Clarke's dissertation [12] or Joline [38], can be used to temporarily export objects inside an aggregate context past their ombudsmen, but only for the duration of a method call.

When combining ombudsmen with an ownership-based effect system, such as the one in Joe_1 or our extension of Joëlle, the obvious question arises, how to report an effect to an object in the shared aggregate? The answer to the question is to externally report effects under the shared aggregate as effects to `owner`. This is imprecise, as not all objects in the `owner` context may access the aggregate. Without additional machinery, like path dependent-types (see *e.g.,* [13]), regions (see *e.g.,* [24]), linearity (see *e.g.,* [10]) or data groups [26], distinguishing ombudsmen for different aggregates is in any case impossible, so the subsuming `aggregate` into `owner` for effects is required for soundness.

6 Concluding Remarks

We have presented an extension to Clarkean ownership types that slightly relaxes owners-as-dominators to enable multiple entry points into a single aggregate. Our extension works well with existing deep ownership systems, and only requires two additional ownership contexts, `aggregate` and `bridge`, and minor extensions to existing type rules. In terms of increasing complexity for the programmer, we believe that multiple contexts of an object does not overly complicate programming, especially since the contexts are limited to two, and the notion of owner specialization, `bridge` is a subset of `owner`, should be as straightforward as any simple notion of regions.

We have implemented the ombudsman system as part of our extended Joëlle compiler on top of JastAddJ [20]. It currently supports deep ownership types, external uniqueness, and a complete implementation of ombudsmen, including staged aggregate construction with ownership transfer.

Acknowledgments. We thank the anonymous reviewers at IWACO 2011 and the ECOOP 2012 reviewers for their valuable feedback which helped us improve our presentation, and correct mistakes.

References

1. Aldrich, J., Chambers, C.: Ownership Domains: Separating Aliasing Policy from Mechanism. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
2. Aldrich, J., Kostadinov, V., Chambers, C.: Alias Annotations for Program Understanding. In: OOPSLA (November 2002)
3. Banerjee, A., Naumann, D.A.: Secure Information Flow and Pointer Confinement in a Java-like Language. In: Proceedings of the Fifteenth IEEE Computer Security Foundations Workshop (CSFW), pp. 253–267. IEEE Computer Society Press (June 2002)
4. Barnett, M., Naumann, J.D.A.: Friends Need a Bit More: Maintaining Invariants Over Shared State. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
5. Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J., Houston, K.A.: Object-oriented analysis and design with applications, third edition. SIGSOFT Softw. Eng. Notes, 33, 11:29–11:29 (2008)
6. Boyapati, C., Lee, R., Rinard, M.: Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In: OOPSLA (2002)
7. Boyapati, C., Liskov, B., Shrira, L.: Ownership Types for Object Encapsulation. In: POPL (2003)
8. Boyapati, C., Liskov, B., Shrira, L., Moh, C.-H., Richman, S.: Lazy Modular Upgrades in Persistent Object Stores. In: OOPSLA, pp. 403–417. ACM, New York (2003)
9. Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In: PLDI (June 2003)
10. Boyland, J., Retert, W., Zhao, Y.: Iterators can be Independent from Their Collections. In: IWACO (2007)
11. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple Ownership. In: OOPSLA (2007)
12. Clarke, D.: Object Ownership and Containment. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia (2001)
13. Clarke, D., Drossopolou, S.: Ownership, Encapsulation and the Disjointness of Type and Effect. In: OOPSLA (2002)
14. Clarke, D., Wrigstad, T.: External Uniqueness is Unique Enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 176–201. Springer, Heidelberg (2003)
15. Clarke, D., Wrigstad, T., Östlund, J., Johnsen, E.B.: Minimal Ownership for Active Objects. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 139–154. Springer, Heidelberg (2008)
16. Clarke, D.G., Potter, J., Noble, J.: Ownership Types for Flexible Alias Protection. In: OOPSLA, pp. 48–64 (1998)
17. Cunningham, D., Drossopoulou, S., Eisenbach, S.: Universe Types for Race Safety. In: VAMP 2007, pp. 20–51 (September 2007)
18. Dietl, W.: Universe Types: Topology, Encapsulation, Genericity, and Tools. Ph.D., Department of Computer Science, ETH Zurich, Doctoral Thesis ETH No. 18522 (December 2009)
19. Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 28–53. Springer, Heidelberg (2007)
20. Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: OOPSLA, pp. 1–18. ACM, New York (2007)

21. Ernst, E., Ostermann, K., Cook, W.R.: A Virtual Class Calculus. In: Proceedings of Principles of Programming Languages (POPL) (January 2006)
22. Fahndrich, M., Xia, S.: Establishing Object Invariants with Delayed Types. SIGPLAN Not. 42(10), 337–350 (2007)
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
24. Greenhouse, A., Boyland, J.: An Object-Oriented Effects System. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 205–229. Springer, Heidelberg (1999)
25. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (2001)
26. Leino, K.R.M.: Data groups: specifying the modification of extended state. In: OOPSLA, pp. 144–153. ACM, New York (1998)
27. Leino, K.R.M., Müller, P.: Object Invariants in Dynamic Contexts. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
28. Li, P., Cameron, N., Noble, J.: Mojojojo - more ownership for multiple owners. In: International Workshop on Foundations of Object-Oriented Languages, FOOL (2010)
29. Lu, Y., Potter, J., Xue, J.: Ownership Downgrading for Ownership Types. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 144–160. Springer, Heidelberg (2009)
30. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for controlling representation exposure. In: Poetzsch-Heffter, A., Meyer, J. (eds.) Programming Languages and Fundamentals of Programming, pp. 131–140. Technical Report 263, Fernuniversität Hagen (1999)
31. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular Specification of Frame Properties in JML. In: Concurrency and Computation Practice and Experience (2003)
32. Müller, P., Rudich, A.: Ownership Transfer in Universe Types. In: OOPSLA, pp. 461–478. ACM, New York (2007)
33. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
34. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, Uniqueness, and Immutability. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 178–197. Springer, Heidelberg (2008)
35. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic Ownership for Generic Java. In: OOPSLA, pp. 311–324. ACM, New York (2006)
36. Schäfer, J., Poetzsch-Heffter, A.: CoBoxes: Unifying Active Objects and Structured Heaps. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 201–219. Springer, Heidelberg (2008)
37. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
38. Wrigstad, T.: Ownership-Based Alias Management. PhD thesis, Royal Institute of Technology, Kista, Stockholm (May 2006)
39. Wrigstad, T., Pizlo, F., Meawad, F., Zhao, L., Vitek, J.: Loci: Simple Thread-Locality for Java. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 445–469. Springer, Heidelberg (2009)
40. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and Immutability in Generic Java. In: OOPSLA 2010, pp. 598–617. ACM, New York (2010)

Inference and Checking of Object Ownership

Wei Huang[1], Werner Dietl[2], Ana Milanova[1], and Michael D. Ernst[2]

[1] Rensselaer Polytechnic Institute
[2] University of Washington

Abstract. Ownership type systems describe a heap topology and enforce an encapsulation discipline; they aid in various program correctness and understanding tasks. However, the annotation overhead of ownership type systems has hindered their widespread use. We present a unified framework for specification, type inference and type checking of ownership type systems, and instantiate the framework for two such systems: Universe Types and Ownership Types. We present an objective metric defining a "best typing" for these type systems, and develop an inference approach that maximizes the metric. The programmer can influence the inference by adding partial annotations to the program. We implemented the approach on top of the Checker Framework and present the results of an experimental evaluation.

1 Introduction

When a type system requires annotations in the source code, the annotation burden on programmers inhibits practical adoption. Therefore, it is important to help programmers transform unannotated or partially-annotated programs to fully-annotated ones. Another benefit of type inference is that it reveals valuable information about how existing programs use the concepts expressed in the type system.

Automatic type inference is especially difficult for type systems that allow multiple valid typings, such as ownership type systems [7]. The notion of the "best typing" is not well-understood or formalized.

This paper presents a unified framework for specifying ownership-like type systems as well as efficient type inference and checking techniques. We give a formal way to define the best typing and design efficient type inference techniques that infer best typings. We have instantiated the framework for two well-known ownership type systems: Universe Types [8], which enforces the *owner-as-modifier* encapsulation discipline, and Ownership Types [4], which enforces the *owner-as-dominator* encapsulation discipline, and present an empirical evaluation.

This paper makes the following contributions:

- A unified framework for specifying the type rules of ownership type systems and instantiations of the framework for two well-known ownership type systems, Universe Types (UT), and Ownership Types (OT). (See Sect. 2.)
- A formalization of the notion of "best typing" for ownership type systems. The programmer specifies a ranking over all valid typings; the highest ranked

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 181–206, 2012.

$$
\begin{array}{lll}
cd & ::= \mathsf{class\ C\ extends\ D\ \{}\ \overline{fd}\ \ \overline{md}\ \mathsf{\}} & \textit{class} \\
fd & ::= \tau\ \mathsf{f} & \textit{field} \\
md & ::= \tau\ \mathsf{m}(\tau\ \mathsf{x})\ \mathsf{\{}\ \overline{\tau\ \mathsf{y}}\ \mathsf{s;\ return\ y\ \}} & \textit{method} \\
\mathsf{s} & ::= \mathsf{s; s} \mid \mathsf{x = new}\ \tau() \mid \mathsf{x = y} \mid \mathsf{x.f = y} \mid \mathsf{this.f = y} & \textit{statement} \\
 & \quad \mid \mathsf{x = y.f} \mid \mathsf{x = this.f} \mid \mathsf{x = y.m(z)} \mid \mathsf{x = this.m(z)} & \\
\tau & ::= q\ \mathsf{C} & \textit{qualified type} \\
q & \in Q & \textit{qualifier}
\end{array}
$$

Fig. 1. Syntax of a core OO language. The set Q of all qualifiers q is a framework parameter instantiated for specific ownership type systems.

typing is the best typing. The ranking is a heuristic reflecting the desire for deep ownership trees — higher ranked (i.e., "better") typings give rise to deeper runtime ownership trees. Deep ownership trees are desirable, because they expose high degree of encapsulation. (See Sect. 3.)

- A unified type inference approach. The inference reflects programmer intent in two ways: (1) it accepts a programmer-specified ranking over typings, which guides the automatic inference towards the best of many valid typings, and (2) it accepts partially-annotated programs and seamlessly integrates programmer-provided annotations with automatic inference: the programmer may choose to annotate a subset of the variables; the automatic inference fills in the rest, guided by the ranking towards the best typing. (See Sect. 4.)
- A formulation of Universe Types inference as an instance of the unified approach. We infer the "best UT typing", in quadratic time, without annotations. (See Sect. 4.3.)
- A demonstration that while the best UT typing is tractable, the best OT typing is challenging. Our approach cannot always infer the best OT typing without annotations. We scale Ownership Type inference by asking the programmer to provide a small number of annotations (6 per kLOC on average). We infer the "best OT typing" for the partially-annotated program in quadratic time. (See Sect. 4.4.)
- An empirical evaluation which presents type inference results for UT and OT on Java programs of up to 110kLOC, and a comparison of UT and OT. (See Sect. 5.)

2 Unified Framework for Ownership Type Systems

This section describes our unified framework for specifying ownership type systems. The framework can be instantiated to specific ownership type systems. Sect. 2.1 describes the framework's unified typing rules, Sect. 2.2 instantiates the framework for Universe Types, and Sect. 2.3 instantiates it for Ownership Types.

For brevity, we restrict our formal attention to a core calculus in the style of Vaziri et al. [26] whose syntax appears in Fig. 1. The language models Java with a syntax in A-normal form. For brevity, we assume in the presentation that all methods have a single parameter; our implementation handles the general case.

2.1 Framework and Unified Typing Rules

The framework is instantiated to a specific type system by defining three framework parameters: (1) the set of type qualifiers Q with the corresponding subtyping hierarchy, (2) the viewpoint adaptation function $\triangleright$ (described below), and (3) type-system-specific constraints $\mathcal{B}$, enforced in addition to the standard subtyping and viewpoint adaptation constraints.

In contrast to a formalization of pure Java, a type τ has two orthogonal components: ownership type qualifier q and Java class type C. The ownership type system is *orthogonal* (i.e., independent) to the Java type system, which allows us to specify typing rules over type qualifiers q alone.

Framework parameter $\triangleright$ defines *viewpoint adaptation* [8]. For example, the type of $\mathsf{x.f}$ is not just the declared type of field f — it is the type of f adapted from the point of view of x. In ownership type systems, viewpoint adaptation adapts the type of a field, formal parameter, or return type, from the viewpoint of the *receiver* at the corresponding field access or method call to the viewpoint of the current object this. Viewpoint adaptation is performed at field accesses and method calls and is written $q \triangleright q'$, which denotes that type q' is adapted from the point of view of type q to the viewpoint of the current object this. Viewpoint adaptation rules for each type system are given in Sections 2.2 and 2.3.

Fig. 2 shows the unified typing rules over the A-normal-form Java syntax. The figure makes use of the three framework parameters. The environment Γ is used to look up the type qualifier of a variable. Rule (TNEW) ensures that the instantiated type is a subtype of the type of the left-hand side and enforces the additional type-system-specific constraints determined by $\mathcal{B}$. Similarly, rule (TASSIGN) checks the types in assignments. The rules in Fig. 2 separate access through the current object this from other accesses. Rule (TWRITE) adapts the type of the field, and creates the subtype constraint between the type on the right-hand-side and the adapted type of f. Auxiliary function $\mathit{typeof}(\mathsf{f})$ retrieves the type of field f from its declaration. Rule (TREAD) ensures that the adapted field type is a subtype of the type of the left-hand-side. Rule (TCALL) uses $\mathit{typeof}(\mathsf{m})$ to retrieve the type of method m, namely $q \rightarrow q'$, from its declaration. Rule (TCALL) then creates the expected subtyping constraint between the type of the actual argument z and the adapted type of the formal parameter, as well as the subtyping constraint between the adapted return type and the type of the left-hand-side x. Finally, rules (TWRITETHIS), (TREADTHIS), and (TCALLTHIS) perform the corresponding operations, without viewpoint adaptation.

We now instantiate the framework for two well-known ownership type systems, Universe Types (UT) [8,5] and Ownership Types (OT) [4]. The framework can also be instantiated for a variety of other ownership-like type systems, including EnerJ [22], a type system for energy efficiency, and AJ [26], a type system for data-centric synchronization.

2.2 Universe Types

Universe Types (UT) [8,5] is a lightweight ownership type system that optionally enforces the *owner-as-modifier* encapsulation discipline. Informally, this means

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TNEW})}(q_{\mathsf{x}}, q)}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}\ (\text{TNEW})$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TASSIGN})}(q_{\mathsf{x}}, q_{\mathsf{y}})}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\text{TASSIGN})$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \triangleright q_{\mathsf{f}} \quad \mathcal{B}_{(\text{TWRITE})}(q_{\mathsf{x}}, q_{\mathsf{f}}, q_{\mathsf{y}})}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ (\text{TWRITE})$$

$$\frac{\mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{f}} \quad \mathcal{B}_{(\text{TWRITETHIS})}(q_{\mathsf{f}}, q_{\mathsf{y}})}{\Gamma \vdash \mathsf{this.f} = \mathsf{y}}\ (\text{TWRITETHIS})$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad q_{\mathsf{y}} \triangleright q_{\mathsf{f}} <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TREAD})}(q_{\mathsf{y}}, q_{\mathsf{f}}, q_{\mathsf{x}})}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\text{TREAD})$$

$$\frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad q_{\mathsf{f}} <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TREADTHIS})}(q_{\mathsf{f}}, q_{\mathsf{x}})}{\Gamma \vdash \mathsf{x} = \mathsf{this.f}}\ (\text{TREADTHIS})$$

$$\frac{\mathit{typeof}(\mathsf{m}) = q \to q' \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \quad q_{\mathsf{z}} <: q_{\mathsf{y}} \triangleright q \quad q_{\mathsf{y}} \triangleright q' <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TCALL})}(\mathsf{m}, q_{\mathsf{y}}, q_{\mathsf{x}})}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}\ (\text{TCALL})$$

$$\frac{\mathit{typeof}(\mathsf{m}) = q \to q' \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \quad q_{\mathsf{z}} <: q \quad q' <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TCALLTHIS})}(\mathsf{m}, q_{\mathsf{x}})}{\Gamma \vdash \mathsf{x} = \mathsf{this.m(z)}}\ (\text{TCALLTHIS})$$

Fig. 2. Unified typing rules. The ownership type system is independent from the Java type system, which allows us to specify the typing rules over qualifiers q alone.

that an object can be modified only by its owner and by its peers, i.e., objects that have the same owner. There are three source-level qualifiers, i.e., $Q_{UT} = \{\mathsf{peer}, \mathsf{rep}, \mathsf{any}\}$:

- peer: an object that is referenced by a peer reference x is part of the same representation as the current object. In other words, the two objects have the same owner.
- rep: an object that is referenced by a rep reference x is part of the current (i.e., this) object's representation. In other words, the current object is the *owner* of the object referenced by x.
- any: the any qualifier does not provide any information about the ownership of the object.

The formalization of Universe Types uses the qualifier lost to express that the result of viewpoint adaptation cannot be expressed statically, that is, a type declaration enforces an ownership constraint, but the constraint is not expressible from the current viewpoint. Qualifier lost is used only internally and users cannot annotate references as lost. In contrast to previous work [5], we type the current object this as peer and use separate rules for accesses through this, instead of adding a self qualifier.

```
1  class XStack {
2    any Link top;
3    XStack() {
4      top = null;
5    }
6    void push(any X d1) {
7      rep Link newTop;
8      newTop = new rep Link(); [l]
9      newTop.init(d1);
10     newTop.next = top;
11     top = newTop;
12   }
13   void main(String[] arg) {
14     rep XStack s;
15     s = new rep XStack(); [s]
16     any X x = new rep X(); [x]
17     s.push(x);
18   }
19 }
20 class Link {
21   any Link next;
22   any X data;
23   void init(any X d2) {
24     next = null;
25     data = d2;
26   }
27 }
```

```
1  class XStack {
2    ⟨rep|p⟩ Link top;
3    XStack() {
4      top = null;
5    }
6    void push(⟨p|p⟩ X d1) {
7      ⟨rep|p⟩ Link newTop;
8      newTop = new ⟨rep|p⟩ Link(); [l]
9      newTop.init(d1);
10     newTop.next = top;
11     top = newTop;
12   }
13   void main(String[] arg) {
14     ⟨rep|rep⟩ XStack s;
15     s = new ⟨rep|rep⟩ XStack(); [s]
16     ⟨rep|rep⟩ X x = new ⟨rep|rep⟩ X(); [x]
17     s.push(x);
18   }
19 }
20 class Link {
21   ⟨own|p⟩ Link next;
22   ⟨p|p⟩ X data;
23   void init(⟨p|p⟩ X d2) {
24     next = null;
25     data = d2;
26   }
27 }
```

Fig. 3. A program with qualifiers for UT (left) and OT (right) as inferred by our tool. The boxed italic letters denote object allocation sites.

The qualifiers form the following subtyping hierarchy:

$$\mathsf{rep} <: \mathsf{lost} \qquad \mathsf{peer} <: \mathsf{lost} \qquad \mathsf{lost} <: \mathsf{any}$$

that is, qualifiers peer and rep are incomparable to each other and are subtypes of lost, and all qualifiers are below any.

Viewpoint adaptation in UT is defined as follows:

$$\begin{array}{lll} \mathsf{peer} \triangleright \mathsf{peer} & = \mathsf{peer} & \\ \mathsf{rep} \triangleright \mathsf{peer} & = \mathsf{rep} & \\ \_ \triangleright \mathsf{any} & = \mathsf{any} & \\ q \triangleright q' & = \mathsf{lost} & \text{otherwise} \end{array}$$

Viewpoint adaptation is applied only when the receiver is not this. The type of the receiver is q_{x} at (TWRITE), q_{y} at (TREAD) and q_{y} at (TCALL). Consider x.f = y. If x is rep, then the current object is the owner of the x object. If the type of f is peer, then the x object and field f object are peers. Therefore, the current object is the

owner of the f object, which is expressed by the fact that the type of f, adapted from the point of view of x's rep, is rep.

UT imposes additional constraints, beyond the standard subtyping and viewpoint adaptation constraints. In our framework, these constraints are expressed by framework parameters $\mathcal{B}$:

$$\begin{aligned}
\mathcal{B}_{(\text{TNEW})}(q_l, q_r) &= \{q_r \neq \mathsf{any}\} \\
\mathcal{B}_{(\text{TWRITE})}(q_r, q_f, q_o) &= \{\underline{q_r \neq \mathsf{any}}, q_r \rhd q_f \neq \mathsf{lost}\} \\
\mathcal{B}_{(\text{TCALL})}(\mathsf{m}, q_r, q_o) &= \text{let } \mathit{typeof}(\mathsf{m}) = q \to q' \text{ in} \\
&\quad \text{if } \mathit{impure}(\mathsf{m}) \text{ then } \{\underline{q_r \neq \mathsf{any}}, q_r \rhd q \neq \mathsf{lost}\} \\
&\quad \text{else } \{q_r \rhd q \neq \mathsf{lost}\}
\end{aligned}$$

The $\mathcal{B}$ sets for (TASSIGN), (TWRITETHIS), (TREAD), (TREADTHIS), and (TCALLTHIS) are all empty; these rules do not impose additional constraints.

In (TNEW), the newly created object needs to be created in a concrete ownership context and therefore needs peer or rep as ownership qualifiers. In (TWRITE), the adapted field type cannot be lost, and in (TCALL), the adapted formal parameter type cannot be lost.

The underlined constraints above enforce the owner-as-modifier encapsulation discipline — they disallow modifications in statically unknown contexts. The receiver cannot be any in (TWRITE) or in (TCALL) if the method is impure, that is, if the method might have nonlocal side effects. We use our method purity inference tool [13], which relies on a type system for reference immutability and is another instantiation of the unified framework described here. Note that, in contrast to other formalizations [7], we do not need to forbid lost as receiver, because our syntax here is in A-normal form and the programmer cannot explicitly write lost.

Fig. 3 (left) shows a program annotated with Universe types. Variable newTop at line 7 and the Link object l are typed rep, meaning that the XStack object is the owner of the Link object. References top (line 2) and next are any because they are never used to modify the object that they refer to. References d1 and d2 are any as well, as they are never used to modify the object they refer to.

Ownership type systems give rise to a hierarchical ownership structure shown with an *ownership tree*. Fig. 4 shows the object graph and the corresponding ownership tree for the program in Fig. 3. *root* is the owner of objects s and x and s is the owner of l.

2.3 Ownership Types

We now consider the classical Ownership Types (OT) [4], restricted to one ownership parameter. The system enforces the *owner-as-dominator* encapsulation discipline, meaning that an object cannot be exposed outside of the boundary of its owner, or in other words, all access paths to the object go through its owner. There are three base ownership modifiers in Ownership Types:

- rep refers to the current object this.
- own refers to the owner of the current object.
- p is an ownership parameter passed to the current object.

Fig. 4. Object graph (left) and ownership tree (right) for the example in Fig. 3. UT and OT give rise to the same ownership tree. In the object graph we show all references between objects. In the ownership tree we draw an arrow from the owned object to its owner and put all objects with the same owner into a dashed box.

OT qualifiers have the form $\langle q_0|q_1\rangle$, where q_0 and q_1 are one of rep, own, or p. A qualifier $\langle q_0|q_1\rangle$ for reference variable x is interpreted as follows. Let i be the object referenced by x. q_0 is the *owner* of i, from the point of view of the current object, and q_1 is the *ownership parameter* of i, again, from the point of view of the current object. Informally, the ownership parameter q_1 refers to an object, which objects referenced by i might use as owner. For example, $\langle$rep|own$\rangle$ x means that the owner of i is the current object this, and the ownership parameter passed to i is the owner of the current object. Transitively, objects referenced by i, for example, from its fields, can have as owner (1) i itself, by using rep, (2) the current object, by using own, or (3) the owner of the current object, by using p.

There are six type qualifiers:
$Q_{OT} = \{\langle$rep|rep$\rangle, \langle$rep|own$\rangle, \langle$rep|p$\rangle, \langle$own|own$\rangle, \langle$own|p$\rangle, \langle$p|p$\rangle\}$, and there is no subtyping hierarchy. The type of this is $\langle$own|p$\rangle$.

Viewpoint adaptation $\triangleright$ is defined as follows:

$$\begin{aligned}\langle q_0|q_1\rangle \triangleright \langle\text{own}|\text{own}\rangle &= \langle q_0|q_0\rangle\\ \langle q_0|q_1\rangle \triangleright \langle\text{own}|\text{p}\rangle &= \langle q_0|q_1\rangle\\ \langle q_0|q_1\rangle \triangleright \langle\text{p}|\text{p}\rangle &= \langle q_1|q_1\rangle\end{aligned}$$

Viewpoint adaptation disallows the adapted type from containing rep, which accounts for the static visibility constraint [4].

As an example, let us discuss the first rule: the adapted type of $\langle$own|own$\rangle$ from the point of view of $\langle q_0|q_1\rangle$ is $\langle q_0|q_0\rangle$. If an object i has type $\langle q_0|q_1\rangle$ from the point of view of the current this object, this means that the owner of i is q_0. If object j has type $\langle$own|own$\rangle$ from the point of view of i, this means that both j's owner and ownership parameter are instantiated to the *owner* of i. Therefore, j will have type $\langle q_0|q_0\rangle$ from the point of view of this.

As in UT, viewpoint adaptation is applied only when the receiver variable is not this. When the receiver is this, there is no need to adapt, as the object remains in the same context, the context of this.

For example, consider a field read x = y.f. Let y have type $\langle$rep|rep$\rangle$ and let field f have type $\langle$own|p$\rangle$. Then y.f has type $\langle$rep|rep$\rangle$. The first rep in this type can be explained as follows: Owner own in the type of f gives us that the owner

of the f object is the same as the owner of the y object, and owner rep in the type of y gives us that the owner of the y object is the current object. Thus, the owner of the f object, from the point of view of the current object, is the current object.

In OT, all $\mathcal{B}$ sets are empty as the system does not impose additional constraints beyond the standard subtyping and viewpoint adaptation constraints. Note that the subtyping constraints degenerate into equality constraints as OT does not have a subtyping hierarchy.

Fig. 3 (right) shows the XStack program annotated with Ownership Types. The XStack object s is ⟨rep|rep⟩ meaning that the owner of s is *root* and the ownership parameter passed to s is *root* as well. The Link object l is ⟨rep|p⟩ meaning that the enclosing XStack object is the owner of l, and the ownership parameter of the XStack object is passed to l as an ownership parameter. Variable next (line 21) has type ⟨own|p⟩ which means that the next link and the current link have the same owner, the enclosing XStack object. data is typed ⟨p|p⟩ meaning that its owner is the ownership parameter of Link which resolves to *root*. The resulting ownership tree is shown in Fig. 4. Note that for this program UT and OT give rise to the same ownership tree. In general however, UT and OT capture different ownership structure, as we will discuss in Sect. 5.

We conclude this section with a brief discussion of why we choose to restrict OT to one ownership parameter. As an experiment, we instantiated the unified framework for ownership type systems with 2 and 3 ownership parameters. However, the complexity of annotations was so overwhelming that we could not manually verify the inferred results. We concluded that in order to use Ownership Types in practice, we must restrict the system to one ownership parameter.

3 Heuristic Ranking over Typings

Ownership type systems typically allow many different typings for a given program. The trivial typings that apply to every program (peer in Universe Types, or ⟨p|p⟩ in Ownership Types) give rise to flat ownership trees where every object is a child of *root*. These typings permit every access and modification, so they do not express the programmer's intent nor detect/prevent coding errors. These goals are better served by inferring deep ownership trees, not trivial flat trees.

This section formalizes the notion of the best typing using a *ranking over all typings*. For ownership types, the ranking is a heuristic/proxy for deep ownership trees — a higher ranked typing would likely give rise to a deeper (i.e., better) runtime ownership tree than a lower ranked typing.

We begin by defining the notion of a valid typing. Let P be a program and F be an ownership type system with universal set of qualifiers Q_F. A *typing* $T_{P,F}$ is a mapping from the variables[1] in P to the type qualifiers in Q_F. A typing $T_{P,F}$ is a *valid typing for P in F* when it renders P well-typed in F. Note that

[1] For the rest of the paper we use "variables" to denote all annotatable types, that is, local variable, parameter, return, allocation site, and field types.

a valid typing $T_{P,F}$ must maintain programmer-provided annotations in P, that is, if a variable v is annotated by the programmer with q, then for every valid typing $T_{P,F}$, we have $T_{P,F}(v) = q$.

We proceed to define an objective function o that can be used to rank valid typings, and instantiations for UT and OT. The objective function o takes a valid typing T and returns a tuple of numbers[2]. The tuples are ordered lexicographically.

To create the tuple, the objective function o assumes that the qualifiers are partitioned and the partitions are ordered. Then, each element of the tuple is the number of variables in T whose type is in the corresponding partition.

3.1 Objective Function for Universe Types

For UT, the function is instantiated as

$$o_{UT}(T) = (|T^{-1}(\mathsf{any})|, |T^{-1}(\mathsf{rep})|, |T^{-1}(\mathsf{peer})|)$$

The partitioning and ordering is

$$\{\mathsf{any}\} > \{\mathsf{rep}\} > \{\mathsf{peer}\}$$

Each qualifier falls in its own partition. This means, informally, that we prefer any over rep and peer, and rep over peer. More formally, the partitioning and ordering gives rise to a preference ranking O_{UT} over all qualifiers:

$$O_{UT} : \mathsf{any} > \mathsf{rep} > \mathsf{peer}$$

Note that this preference ranking is not related to subtyping. We have $T_1 > T_2$ iff T_1 has a larger number of variables typed any than T_2, or T_1 and T_2 have the same number of any variables, but T_1 has a larger number of rep variables than T_2. Function o_{UT} gives a natural ranking over the set of valid typings for UT. In fact, the maximal (i.e., best) typing according to the above ranking, *maximizes the number of allocation sites typed* rep, which is a good proxy for a deep UT ownership tree.

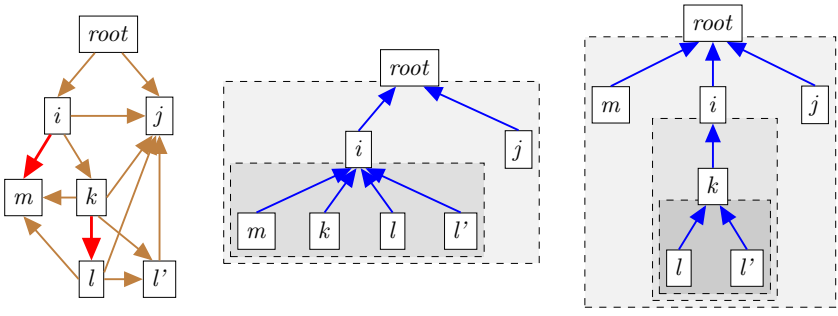
It is interesting to note that an o with exactly one qualifier per partition gives a meaningful heuristic ranking for other type systems, most notably reference immutability [13,25] and AJ [26].

3.2 Objective Function for Ownership Types

OT cannot use an objective function with one qualifier per partition. Informally, the base modifiers are preference-ranked as

$$\mathsf{rep} > \mathsf{own} > \mathsf{p}$$

[2] Strictly, o and T are defined in terms of a specific type system F and program P; for brevity, we omit the subscripts when they are clear from context.



(a) Object graph (b) OT tree for T_1 (c) OT tree for T_2

Fig. 5. Ownership trees resulting from typings T_1 and T_2. Edges $i \to m$ and $k \to l$ (shown in red in the object graph) cannot be typed with owner rep simultaneously.

but, say, ⟨rep|rep⟩ should not carry more weight than ⟨rep|p⟩. The objective function should maximize the number of rep owners *regardless of ownership parameters*.

To illustrate this point, suppose that qualifiers $\langle q_0|q_1\rangle$ were ordered lexicographically based on the ranking of base modifiers, and consider Fig. 5. A variable roughly corresponds to an edge in the object graph [14], and therefore, we use typing of edges instead of typing of variables. Edges $i \to m$ and $k \to l$ cannot be typed with owner rep simultaneously, because of the restriction to one ownership parameter. Thus, one valid typing, call it T_1, types $root \to i$, $root \to j$ and $i \to m$ as ⟨rep|rep⟩, $i \to k$ as ⟨rep|own⟩, and the rest of the edges as either ⟨own|_⟩ or ⟨p|p⟩. T_1 gives rise to the ownership tree in Fig. 5(b); T_1 flattens the tree at l and l' — the owner of l and l' is i, even though k dominates both l and l' and we would like to have k as the owner of l and l'. Another valid typing, call it T_2, types $root \to i$, $root \to j$ as ⟨rep|rep⟩, $i \to k$ as ⟨rep|own⟩, $k \to l$ and $k \to l'$ as ⟨rep|p⟩, and the rest of the edges as either ⟨own|_⟩ or ⟨p|p⟩. T_2 gives rise to the tree in Fig. 5(c); this tree is better than the tree in Fig. 5(b) because it has more dominance. Note that lexicographical ordering ranks T_1 higher than T_2 because it contains 3 ⟨rep|rep⟩ typings, while T_2 contains only 2 ⟨rep|rep⟩ typings. However, T_2 is the better typing, because it contains 5 ⟨rep|_⟩ typings, one more than T_1, and therefore, it preserves more dominance in the ownership tree than T_1.

In OT, valid typings are ranked using the following o:

$$o_{OT}(T) = (|T^{-1}(\langle \mathsf{rep}|\_\rangle)|, |T^{-1}(\langle \mathsf{own}|\_\rangle)|, |T^{-1}(\langle \mathsf{p}|\_\rangle)|)$$

Here $T^{-1}(\langle \mathsf{rep}|\_\rangle)$ is the set of variables typed with owner rep, i.e., typed ⟨rep|rep⟩, ⟨rep|own⟩ or ⟨rep|p⟩. $T^{-1}(\langle \mathsf{own}|\_\rangle)$ is the set of variables typed with owner own, and $T^{-1}(\langle \mathsf{p}|\_\rangle)$ is the set of variables typed with owner p. The primary goal is to maximize the number of variables typed with owner rep (regardless of ownership parameters). Thus, the ranking *maximizes the number of edges in the object graph that are typed* rep, or in other words, the best typing preserves the most dominance (ownership). This is a good proxy for a deep OT ownership tree.

Our type inference approach (Sect. 4) requires that all qualifiers are preference-ranked and the ranking over qualifiers preserves partition ranking. Unlike o_{UT}, o_{OT} does not give rise to such ranking (e.g., $\langle \mathsf{rep}|\mathsf{rep}\rangle$ and $\langle \mathsf{rep}|\mathsf{p}\rangle$ are equally preferred by o_{OT}). We use lexicographical order over the base modifiers:

$$O_{OT} : \langle \mathsf{rep}|\mathsf{rep}\rangle > \langle \mathsf{rep}|\mathsf{own}\rangle > \langle \mathsf{rep}|\mathsf{p}\rangle > \langle \mathsf{own}|\mathsf{own}\rangle > \langle \mathsf{own}|\mathsf{p}\rangle > \langle \mathsf{p}|\mathsf{p}\rangle$$

O_{OT} preserves the partition ranking (e.g., $\langle \mathsf{rep}|\mathsf{p}\rangle > \langle \mathsf{own}|\mathsf{own}\rangle$) and preference-ranks qualifiers within partitions (e.g., $\langle \mathsf{rep}|\mathsf{rep}\rangle > \langle \mathsf{rep}|\mathsf{own}\rangle > \langle \mathsf{rep}|\mathsf{p}\rangle$).

3.3 Maximal Typing

A *maximal typing* is a typing that maximizes o (i.e., the best typing(s) according to the heuristics encoded in o).

Definition 1. Maximal Typing. Given an objective function o over the set of valid typings, a valid typing T is a *maximal typing of P in F* under o, if for every valid typing T', we have $T' \neq T \Rightarrow T \geq T'$.

Perhaps somewhat unexpectedly, for UT, as well as other interesting systems such as reference immutability [13], there exists a *unique maximal typing*. This is discussed in detail in the next section. For OT however, in general, there are multiple maximal typings, i.e., there are multiple typings that maximize o_{OT}. Consider the following program:

```
1  x = new X(); [x]
2  y = new Y(); [y]
3  x.f = y;
```

There are variables x, y, field f, and allocation sites x and y. Typing T_1 types the program as follows: $T_1(\mathsf{x}) = T_1(x) = \langle \mathsf{rep}|\mathsf{own}\rangle$, $T_1(\mathsf{y}) = T_1(y) = \langle \mathsf{rep}|\mathsf{own}\rangle$, and $T_1(\mathsf{f}) = \langle \mathsf{own}|\mathsf{p}\rangle$. Typing T_2 types the program as follows:. $T_2(\mathsf{x}) = T_2(x) = \langle \mathsf{rep}|\mathsf{rep}\rangle$, $T_2(\mathsf{y}) = T_2(y) = \langle \mathsf{rep}|\mathsf{rep}\rangle$, and $T_2(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle$. Clearly, $o_{OT}(T_1) = o_{OT}(T_2) = (4, 1, 0)$. There are other valid typings that maximize o_{OT} as well. There are nontrivial examples as well.

The following section describes a unified type inference approach, which can be used to compute the unique maximal typing for UT, and a maximal typing for OT given user annotations.

4 Unified Type Inference

The unified inference and checking system works on completely unannotated programs, as well as on partially-annotated programs. We believe that neither fully automatic inference nor fully manually annotated programs are feasible choices. In many interesting systems, fully automatic inference is impossible; that is, the programmer must provide initial annotations which typically reflect semantics that is impossible to infer. We envision a cooperative system that fills in as many annotations as possible and queries the programmer for a small set of

annotations on certain variables to resolve ambiguities. The system seamlessly integrates programmer-provided annotations with inferred annotations.

The key idea in our system is to compute a set-based solution S instead of a single typing. S maps variables to *sets* of qualifiers: for every statement s, for every variable v in s, and for every qualifier $q \in S(v)$, there are qualifiers in the sets of the remaining variables in s, such that q and those qualifiers make statement s type check. Interestingly, for some systems such as UT, the set-based solution S, which is inexpensive to compute, implies the unique maximal typing. For other systems, such as OT, where the unique maximal typing does not exist, S pinpoints the places where programmer-provided annotations must be added, and as a result, reduces the number of manual annotations significantly.

Sect. 4.1 describes the computation of the set-based solution S and Sect. 4.2 describes its properties. Then, Sects. 4.3 and 4.4 describe how the type inference is instantiated for the two ownership type systems in our study.

4.1 Set-Based Solution

Set Mapping. S maps each program variable (annotatable reference) to a *set* of possible type qualifiers. We fix the program P and type system F, and we write S instead of $S_{P,F}$ for brevity.

The initial mapping, S_0, is defined as follows. Programmer-annotated variables are initialized to the singleton set which contains only the programmer-provided annotation. Variables that are not annotated are initialized to the *maximal set* of qualifiers Q_F. The analysis, a fixpoint iteration, iterates over the statements in the program and refines the initial sets, until it reaches the fixpoint.

Transfer Functions. We now describe the transfer functions applied by fixpoint iteration. There is a transfer function f_s for each statement s. Statements s can be of kinds as shown in Fig. 2. Each f_s takes as input mapping S and outputs an updated mapping S'. Informally, f_s removes all infeasible qualifiers from the sets of the variables $v \in s$. After the application of f_s, for each variable $v_i \in s$ and each $q_i \in S'(v_i)$, there exist $q_1 \in S'(v_1), \ldots, q_{i-1} \in S'(v_{i-1}), q_{i+1} \in S'(v_{i+1}), \ldots, q_k \in S'(v_k)$, such that $q_1, \ldots, q_k$ type check with the rule for s in Fig. 2. The transfer functions are defined in terms of the typing rules in Fig. 2; making s type check requires that the subtyping, viewpoint adaptation, and $\mathcal{B}$ constraints for s hold.

More formally $f_s \colon S \to S'$ is defined as follows:

$$
\begin{array}{l}
\text{foreach } v_i \in s \\
\quad S'(v_i) = \{\, q_i \mid q_i \in S(v_i) \text{ and} \\
\qquad \exists q_1 \in S(v_1), \ldots, q_{i-1} \in S(v_{i-1}), q_{i+1} \in S(v_{i+1}), \ldots, q_k \in S(v_k) \\
\qquad\quad \text{s.t. } q_1, \ldots, q_k \text{ type check with the rule for } s \text{ in Fig. 2} \,\}
\end{array}
$$

For example, the transfer function $f_{\mathsf{x}=\mathsf{y}} : S \to S'$ for UT is as follows:

$$
\begin{array}{l}
S'(\mathsf{x}) = \{\, q \mid q \in S(\mathsf{x}) \text{ and } \exists q_{\mathsf{y}} \in S(\mathsf{y}) \text{ s.t. } q_{\mathsf{y}} <: q \,\} \\
S'(\mathsf{y}) = \{\, q \mid q \in S(\mathsf{y}) \text{ and } \exists q_{\mathsf{x}} \in S(\mathsf{x}) \text{ s.t. } q <: q_{\mathsf{x}} \,\}
\end{array}
$$

Suppose that we apply transfer function $f_{\mathsf{x=y}}$ for UT on S, where $S(\mathsf{x}) = \{\mathsf{rep}, \mathsf{peer}\}$ and $S(\mathsf{y}) = \{\mathsf{any}, \mathsf{peer}\}$. $f_{\mathsf{x=y}}$ removes rep from $S(\mathsf{x})$ because there does not exist $q_{\mathsf{y}} \in S(\mathsf{y})$ that will make the type constraint for (TASSIGN), namely $q_{\mathsf{y}} <: \mathsf{rep}$, hold. Next, it removes any from $S(\mathsf{y})$ because $\mathsf{any} <: \mathsf{peer}$ does not hold. After the application of the transfer function, $S'(\mathsf{x}) = \{\mathsf{peer}\}$ and $S'(\mathsf{y}) = \{\mathsf{peer}\}$.

As another example, consider $f_{\mathsf{x.f=y}}$ for OT applied on S, where $S(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$, the set for field f, $S(\mathsf{f}) = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ and $S(\mathsf{y}) = \{\langle \mathsf{own}|\mathsf{own}\rangle\}$. $\langle \mathsf{rep}|\mathsf{rep}\rangle$ is removed from $S(\mathsf{x})$ because there does not exist $q \in S(\mathsf{f})$ such that the type constraint for (TWRITE), namely $\langle \mathsf{rep}|\mathsf{rep}\rangle \triangleright q = \langle \mathsf{own}|\mathsf{own}\rangle$, holds. $\langle \mathsf{rep}|\mathsf{p}\rangle$ and $\langle \mathsf{p}|\mathsf{p}\rangle$ are removed as well, and $S'(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle\}$. Similarly, $\langle \mathsf{rep}|\mathsf{rep}\rangle$, $\langle \mathsf{rep}|\mathsf{own}\rangle$, and $\langle \mathsf{rep}|\mathsf{p}\rangle$ are removed from $S(\mathsf{f})$ (recall that viewpoint adaptation for OT disallows exposed fields from being rep). Thus, $S'(\mathsf{f}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ and $S'(\mathsf{y})$ remains the same.

Fixpoint Iteration. The analysis is a fixpoint iteration. It initializes the mapping S_0 as described earlier in this section, and keeps iterating over the program statements, using the above transfer functions, until one of the following happens: (1) S reaches the fixpoint, i.e., S remains unchanged from the previous iteration, in which case the analysis terminates successfully, or (2) a key is assigned the empty set, in which case the analysis terminates indicating the program is untypable.

The computation fits the requirements of a monotone framework [19]. The property space is the standard lattice of subsets, with the set of qualifiers Q_F being the bottom 0, and the empty set $\emptyset$ being the top 1 of the lattice. The transfer functions are monotone. Therefore, the set-based solution S produced by fixpoint iteration is the unique least solution (for historical reasons sometimes this solution is referred to as the "maximal fixpoint solution" [19]).

4.2 Properties of the Set-Based Solution

Let us now consider the properties of the set-based solution S. These properties help establish that for certain type systems one can derive a maximal (i.e., best) typing from the set-based solution S.

The first proposition states that if the algorithm removes a qualifier q from the set $S(v)$ for variable v, then there does not exist a valid typing that maps v to q. The notation $T \in S_0$ denotes that for every variable v we have $T(v) \in S_0(v)$.

Proposition 1. *Let S be the set-based solution. Let v be any variable in P and let q be any qualifier in F. If $q \notin S(v)$ then there does not exist a valid typing $T \in S_0$, such that $T(v) = q$.*

Proof. (Sketch) We say that q is a valid qualifier for v if there exists a valid typing T, where $T(v) = q$. Let v be the *first* variable that has a valid qualifier q removed from its set $S(v)$ and let f_s be the transfer function that performs the removal. Since q is a valid qualifier there exist valid qualifiers $q_1, ..., q_k$ that make s type check. If $q_1 \in S(v_1)$ and $q_2 \in S(v_2)$, ..., and $q_k \in S(v_k)$, then by

definition, f_s would not have had q removed from $S(v)$. Thus, one of $v_1, \ldots, v_k$ must have had a valid qualifier removed from its set *before* the application of f_s. This contradicts the assumption that v is the first variable that has a valid qualifier removed.

The second proposition states that if we map every variable v to the maximal qualifier in its set $S(v)$ according to its preference ranking over qualifiers[3], and the typing is valid, then this typing maximizes the objective function.

Proposition 2. *Let o be the objective function over valid typings, and S be the set-based solution. The* maximal typing *T is the following: $T(v) = max(S(v))$ for every variable v in P. If T is a valid typing, then T is a maximal typing of P in F under o.*

Proof. (Sketch) We show that T is a maximal typing. Suppose that there exists a valid typing $T' > T$. Let p_i be the most-preferred partition such that $T'^{-1}(p_i) \neq T^{-1}(p_i)$. Since $T' > T$, there must exist a variable v such that $T'(v) = q' \in p_i$, but $T(v) = q \notin p_i$. In other words, T' types v with $T'(v) = q' \in p_i$, but T types v differently — and lesser in the preference ranking, because $T'^{-1}(p_k) = T^{-1}(p_k)$ for $0 \leq k < i$ (here p_k are the more-preferred partitions than p_i). Since $T(v) = max(S(v))$, it follows that $q' \notin S(v)$. By Proposition 1, if $q' \notin S(v)$ there does not exist a valid typing which maps v to q', which contradicts the assumption that T' is a valid typing.

When each partition in the preference ranking has only a single element, then the weaker assumption "there exists a valid typing $T' \geq T$" can be contradicted, showing that the maximal typing is unique.

The *optimality property* holds for a type system F and a program P if and only if the typing derived from the set-based solution S by typing each variable with the maximally/preferred qualifier from its set, is a valid typing.

Property 1. Optimality Property. Let F be a type system augmented with objective function o and let P be a program. The optimality property holds for F and P iff $T(v) = max(S(v))$, for all variables v, is a valid typing.

The set-based solution is computed in $O(n^2)$ time where n is the size of the program. At each iteration through the program, at least one of the $O(n)$ variables changes its set to a smaller set. Therefore, there are at most $O(|Q_F| * n)$ iterations. At each iteration, the computation goes through $O(n)$ statements. Since $|Q_F|$ is a small constant (3 in UT, 6 in OT), it follows that the complexity is $O(n^2)$. Therefore, for type systems for which the optimality property holds for arbitrarily annotated programs, a maximal typing can be computed in quadratic time, with no manual annotations. If the programmer provides inconsistent initial annotations in P, the computation would terminate within $O(n^2)$ time with a message that there is no valid typing for P.

Remarkably, for several interesting systems (UT, AJ, reference immutability), the optimality property holds for unannotated programs, which means that the

[3] Rankings O_{UT} and O_{OT} ensure that the maximal qualifier is uniquely defined.

Variable	Initial	Iteration 1	Iteration 2
top	all	all	all
d1	all	any, peer	any, peer
newTop	all	rep, peer	rep, peer
new Link()	all	rep, peer	rep, peer
s	all	rep, peer	rep, peer
new XStack()	all	rep, peer	rep, peer
x	all	all	all
new X()	all	rep, peer	rep, peer
next	all	any, peer	any, peer
data	all	any, peer	any, peer
d2	all	any, peer	any, peer

Fig. 6. Inference of Universe Types for the example in Fig. 3

unique maximal typing can be computed in $O(n^2)$ time with no manual annotations. However, for OT, the property does not hold for unannotated programs. We will discuss each ownership system in turn in the next two subsections.

4.3 Inference of Universe Types

For Universe Types, the preference ranking over all qualifiers is O_{UT} (previously defined in Sect. 3). Libraries receive default type {peer}.

Fig. 6 illustrates the computation of the set-based solution for UT for the example in Fig. 3. Consider statement s.push(x) at line 17. Initially, $S(\mathsf{s}) = S(\mathsf{x}) = S(\mathsf{d1}) = \{\mathsf{any}, \mathsf{rep}, \mathsf{peer}\}$. In iteration 1, the transfer function for s.push(x) removes any from $S(\mathsf{s})$ because push is impure. It also removes rep from $S(\mathsf{d1})$ because $q \triangleright \mathsf{rep} = \mathsf{lost}$ which the type rule for (TCALL) forbids. See Fig. 6. Choosing the maximal type from each set gives us $T(\mathsf{s}) = \mathsf{rep}$, $T(\mathsf{x}) = \mathsf{any}$, and $T(\mathsf{d1}) = \mathsf{any}$, which type checks with the rule for (TCALL).

We show through case analysis that for each statement s, after the application of the transfer function for s, s type checks with the maximal typing:

(TASSIGN) Consider x = y. We must show that after the application of $f_{\mathsf{x=y}}$, x = y type checks with $max(S'(\mathsf{x}))$ and $max(S'(\mathsf{y}))$.
- If $max(S'(\mathsf{x})) = \mathsf{any}$ the statement type checks with any value for $max(S'(\mathsf{y}))$.
- Suppose that $max(S'(\mathsf{x})) = \mathsf{rep}$. Thus, any is not in $S'(\mathsf{x})$, and therefore any cannot be in $S'(\mathsf{y})$. $max(S'(\mathsf{y}))$ cannot be peer; this contradicts the assumption that $max(S'(\mathsf{x})) = \mathsf{rep}$ (rep would have been removed from x's set). Thus, $max(S'(\mathsf{y})) = \mathsf{rep}$ and x = y type checks.
- Suppose now that $max(S'(\mathsf{x})) = \mathsf{peer}$. The only possible value for $max(S'(\mathsf{y}))$ is peer and the statement again type checks.

(TNEW) is shown exactly the same way.

(TREAD) Consider x = y.f. We must show that after the application of the transfer function $f_{\mathsf{x=y.f}}$, the statement will type check with $max(S'(\mathsf{x}))$, $max(S'(\mathsf{f}))$ and $max(S'(\mathsf{y}))$.

Variable	Initial	Iteration 1	Iteration 2	Iteration 3
`top`	all	all	⟨rep|p⟩	⟨rep|p⟩
`d1`	all	⟨p|p⟩	⟨p|p⟩	⟨p|p⟩
`newTop`	all	⟨rep|p⟩	⟨rep|p⟩	⟨rep|p⟩
`new Link()`	⟨rep|p⟩	⟨rep|p⟩	⟨rep|p⟩	⟨rep|p⟩
`s`	all	all	all	all
`new XStack()`	all	all	all	all
`x`	all	all	all	all
`new X()`	all	all	all	all
`next`	all	⟨own|own⟩, ⟨own|p⟩, ⟨p|p⟩	⟨own|p⟩	⟨own|p⟩
`data`	all	⟨own|own⟩, ⟨own|p⟩, ⟨p|p⟩	⟨p|p⟩	⟨p|p⟩
`d2`	all	⟨own|own⟩, ⟨own|p⟩, ⟨p|p⟩	⟨p|p⟩	⟨p|p⟩

Fig. 7. Inference of Ownership Types for the example in Fig. 3

- If $max(S'(\mathsf{x})) = \mathsf{any}$ this is clearly true.
- Suppose that $max(S'(\mathsf{x})) = \mathsf{rep}$; then $max(S'(\mathsf{f}))$ must be peer and $max(S'(\mathsf{y}))$ must be rep.
- Finally, when $max(S'(\mathsf{x})) = \mathsf{peer}$, one can easily see that $max(S'(\mathsf{f}))$ must be peer and $max(S'(\mathsf{y}))$ must be peer as well.

(TWRITE) and (TCALL) are analogous; they are omitted for brevity. We implemented an independent type checker which verifies the inferred solution.

4.4 Inference of Ownership Types

For Ownership Types, the preference ranking over all qualifiers is O_{UT} (see Sect. 3). Library variables receive default $\{\langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ as explained in [12].

Fig. 7 shows the computation of the set-based solution for the example program in Fig. 3. Note that this computation assumes annotation $\langle \mathsf{rep}|\mathsf{p}\rangle$ at allocation site new Link(); given this annotation, the optimality property holds, and the set-based solution computes the maximal typing for the program.

As mentioned earlier, the optimality property does not always hold in OT. As an example, consider the program:

```
1  x = new A();
2  y = new ⟨own|own⟩ C();
3  x.f = y;
```

The application of transfer functions yields $S(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle\}$, $S(\mathsf{f}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ and $S(\mathsf{y}) = \{\langle \mathsf{own}|\mathsf{own}\rangle\}$. If we map every variable to the maximal qualifier we have

$$T(\mathsf{x}) = \langle \mathsf{rep}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

which fails to type check because $\langle \mathsf{rep}|\mathsf{own}\rangle \triangleright \langle \mathsf{own}|\mathsf{own}\rangle$ equals $\langle \mathsf{rep}|\mathsf{rep}\rangle$, not $\langle \mathsf{own}|\mathsf{own}\rangle$. The set-based solution contains several valid typings. If we chose the maximal value at x, we will have typing

$$T(\mathsf{x}) = \langle \mathsf{rep}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{p}|\mathsf{p}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

and if we chose the maximal value at f, we will have

$$T(\mathsf{x}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

The set-based solution is valuable for two reasons. First, it restricts the search space significantly. Initially, there are 6 possibilities for each variable and there are n variables, leading to 6^n potential typings. Second, the set-based solution highlights the points of non-determinism where programmer-provided annotations can guide the inference to choose one typing over another. With a small number of programmer-provided annotations, OT inference can scale up to large programs. We explain the process in the remainder of this section.

The points of non-determinism arise at field access and method call statements due to viewpoint adaptation. A statement s is a *conflict* if it does not type check with the maximal assignment derived from the set-based solution. In the example above, statement x.f = y is a conflict, because if we map every variable to the maximal qualifier, the statement fails to type check. Our approach performs the following incremental process. Given a program P, which may be unannotated or partially annotated, the tool runs the set-based solver, and if there are conflicts, these conflicts are printed. The programmer selects a subset of conflicts (usually the first 1 to 5), and for each conflict, annotates variables. Then the programmer runs the set-based solver again. This process continues until a program P' is reached, where the optimality property holds for P'. The solver computes a maximal typing for P'.

In the above example, the solver prints conflict x.f = y and the set-based solution

$$\begin{aligned} S(\mathsf{x}) &= \{\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle\} \\ S(\mathsf{f}) &= \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\} \\ S(\mathsf{y}) &= \{\langle \mathsf{own}|\mathsf{own}\rangle\} \end{aligned}$$

If the programmer chooses to annotate x with $\langle \mathsf{rep}|\mathsf{own}\rangle$, this results in typing

$$T(\mathsf{x}) = \langle \mathsf{rep}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{p}|\mathsf{p}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

and if he/she chooses to annotate f with $\langle \mathsf{own}|\mathsf{own}\rangle$ this results in typing

$$T(\mathsf{x}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

5 Empirical Results

5.1 Implementation

Our inference tool is built on top of the Checker Framework [20,6]. The tool extends the Checker Framework to specify type system constraints and preference ranking over qualifiers; it generates the constraints for the type systems by traversing the AST and it implements the set-based constraint solver described in Sect. 4. The tool is freely available at `http://www.cs.rpi.edu/~huangw5/cf-inference/`.

The constraint solver takes as input a number of constraints and it iteratively refines the sets of valid type qualifiers until it reaches a fixpoint. If conflicts (as defined in Sect. 4.4) occur in the solution, the solver prints all conflicts and prompts the user to solve the conflicts by providing manual annotations.

5.2 Results

Benchmarks. We evaluated our implementation using eight Java programs of up to 110kLOC (see Fig. 8). The analysis processes only application code; libraries are handled using the defaults specified in Sect. 4.3 and Sect. 4.4. The analysis is modular, in the sense that it can analyze whatever code is available, including libraries with no main method.

All evaluations were conducted on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM (all benchmarks run within a memory footprint of 1GB). The software environment consists of JDK 1.6 and GNU/Linux 2.6.38.

Benchmark	#Lines	#Meths	Description
JOlden	6223	326	Benchmark suit of 10 small programs
tinySQL	31980	1597	Database engine
htmlparser	62627	1698	HTML parser
ejc	110822	4734	Compiler of the Eclipse IDE
javad	4207	140	Java class file disassembler
SPECjbb	12076	529	SPEC's benchmark for evaluating server side Java
jdepend	4351	328	Java package dependency analyzer
classycle	8972	440	Java class and package dependency analyzer

Fig. 8. The benchmark programs used in our evaluation

Universe Types. Inference of Universe Types requires information about method side effects. As stated earlier, we used our purity inference tool [13]. The purity inference relies on a type system for reference immutability, which itself instantiates our unified framework. The optimality property holds for unannotated programs for UT, and the set-based solver infers the unique maximal typing.

Fig. 9 shows the inference results for Universe Types. Across all benchmarks, 9%–33% of all variables are inferred as any, the best qualifier. 1% to 10% of all variables are inferred as rep. A relatively large percentage (57%–92%) of the variables are inferred as peer, resulting in a flat ownership structure. This is consistent with previous results [7]. There are several possible reasons that lead to flat ownership structures. One is due to utility methods whose formal parameters are passed to impure methods. This forces the formal parameters to be peer. Another reason is that the inference uses the default peer annotation for libraries.

Compared to previous results [7], our inference reports a larger percentage of any variables. One reason is that there are more pure methods in our inference than in [7]. In our inference, pure methods are inferred automatically while in [7] pure methods are annotated manually. For example in javad, 40 methods

Benchmark	#Pure	#Ref	#any	#rep	#peer	#Manual	Time
JOlden	175	685	227 (33%)	71 (10%)	387 (56%)	0	11.3
tinySQL	965	2711	630 (23%)	104 (4%)	1977 (73%)	0	18.2
htmlparser	642	3269	426 (13%)	153 (5%)	2690 (82%)	0	22.9
ejc	1701	10957	1897 (17%)	122 (1%)	8938 (82%)	0	119.7
javad	60	249	31 (12%)	11 (4%)	207 (83%)	0	4.1
SPECjbb	195	1066	295 (28%)	74 (7%)	697 (65%)	0	13.6
jdepend	102	542	95 (18%)	14 (3%)	433 (80%)	0	7.2
classycle	260	946	87 (9%)	11 (1%)	848 (90%)	0	9.9

Fig. 9. The inference results for Universe Types. Column #Ref gives the total number of references excluding implicit parameters this. Column #Pure gives the number of pure methods inferred automatically based on reference immutability [13]. Columns #any, #rep, and #peer give the number of references inferred as any, rep, and peer, respectively. No user annotations are needed for the inference of Universe Types; therefore, there are only zeros in the #Manual column. Last column Time shows the total running time in seconds including parsing the source code, type inference, and type checking.

were manually annotated as pure in [7] while 60 were inferred automatically in our inference; we verified that the extra 20 methods were indeed pure. Another reason is that our qualifier ranking always prefers any over rep. When a variable is mapped to set {any, rep} in the set-based solution, our tool picks any instead of rep. This happens for variable x in Fig. 6. Although x is assigned by a rep allocation site, the tool still infers x as any because x is readonly in the main method. In contrast, Dietl et al. [7] use a different heuristic which uses program location to preference-rank qualifiers. They choose rep over any in certain cases, which results in a larger percentage of variables reported as rep. It is important to note that the larger percentage of any variables *does not imply a flatter ownership tree* compared to [7]; this is because an any variable can refer to a rep object as is the case with variable x. What matters for ownership structure are the allocation sites, and as we shall see shortly, the inference reports a considerably larger percentage of reps for allocation sites compared to reference variables.

Ownership Types. In OT, we add an additional modifier norep, which refers to *root*, as described in detail in [12]. We use norep as the default type for String and boxed primitives such as Boolean, Integer, etc.

Fig. 10 shows the inference results for OT. Note that there are many ⟨norep|_⟩ variables; the majority of these are strings and boxed primitives, e.g. 521 out of 688 ⟨norep|norep⟩ variables in SPECjbb are strings and boxed primitives whose default type is norep.

Compared to UT, a relatively large percentage (4%–24%) of variables are inferred as ⟨rep|_⟩ in OT. Note however, that this *does not imply a deeper ownership tree compared to UT*. In UT, many of the any variables can refer to a rep object (as UT distinguishes readonly access); in contrast, in OT only a rep variable can refer to a rep object. Due to the fact that the optimality property does not hold for OT, as discussed in Sect. 4.4, the inference requires manual annotations.

Benchmark	#Ref	#⟨rep|_⟩	#⟨own|_⟩	#⟨p|_⟩	#⟨norep|_⟩	#Manual	Time
JOlden	685	67 (10%/**10%**)	497 (73%)	24 (4%)	97 (14%)	13 (2)	10.3
tinySQL	2711	224 (8%/**11%**)	530 (20%)	5 (0%)	1952 (72%)	215 (7)	18.4
htmlparser	3269	330 (10%/**11%**)	629 (19%)	36 (1%)	2274 (70%)	200 (3)	33.6
ejc	10957	467 (4%/ **4%**)	1768 (16%)	50 (0%)	8672 (79%)	592 (5)	122.4
javad	249	44 (18%/**19%**)	27 (11%)	74 (30%)	104 (42%)	46 (10)	5.5
SPECjbb	1066	166 (16%/**16%**)	141 (13%)	71 (7%)	688 (65%)	73 (6)	17.1
jdepend	542	130 (24%/**25%**)	156 (29%)	128 (24%)	128 (24%)	26 (6)	13.7
classycle	946	153 (16%/**20%**)	173 (18%)	28 (3%)	592 (63%)	90 (10)	11.7

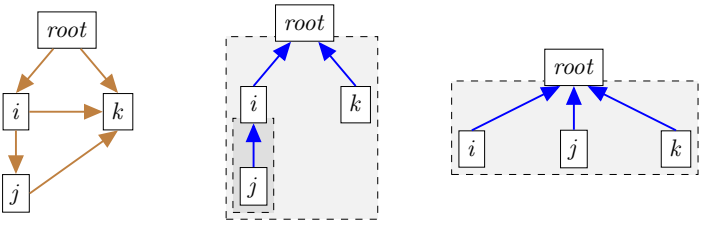
Fig. 10. The inference results for Ownership Types. Column #Ref again gives the total number of references excluding the implicit parameters this. Columns #⟨rep|_⟩, #⟨own|_⟩, #⟨p|_⟩, and #⟨norep|_⟩ give the numbers of variables whose owners are inferred as rep, own, p, and norep, respectively. The boldfaced number in parentheses in column #⟨rep|_⟩ is an upper bound on rep typings; it is discussed in the text. #Manual shows the total number of manual annotations and, in parentheses, the number of annotations per 1kLOC. Time shows the running time in seconds.

Column #Manual gives the total numbers of manual annotations that were added and, in parentheses, the number of annotations per 1kLOC. The annotation burden is low — on average, 6 annotations per 1kLOC. Although the set-based solver cannot produce a maximal typing automatically, it is quite valuable, because it reduces the burden of annotations on programmers. The set-based solver prints all conflicts and lets the programmer choose an annotation that resolves the conflict in such a way that it reflects their intent. This process continues until all conflicts are resolved. By doing so, the first author annotated JOlden (6223 LOC) in approximately 10 minutes and SPECjbb in approximately 2 hours. The annotations reflect the intent of the first author, but not necessary the intent of the programmers of these benchmarks. Finally, the last column Time shows the time in seconds to do type inference and type checking *after* the manual annotations. It is approximately equal to the initial run that outputs all conflicts and does not include the time to annotate the benchmark.

The boldfaced percentage shown in parentheses in column #⟨rep|_⟩, is the percentage of all references that contain a ⟨rep|rep⟩, ⟨rep|own⟩ or ⟨rep|p⟩ in their set-based solution. This is an upper bound on the possible rep typings: even an ownership type system with many ownership parameters will be unable to type a larger percentage of variables as rep. The fact that the percentage of #⟨rep|_⟩'s in our typing is close to this bound, has two implications: (1) our typing is precise (at least with respect to the heuristic defined in Sect. 3), and (2) one ownership parameter may be sufficient in practice (again, if the goal is to maximize the number of rep typings).

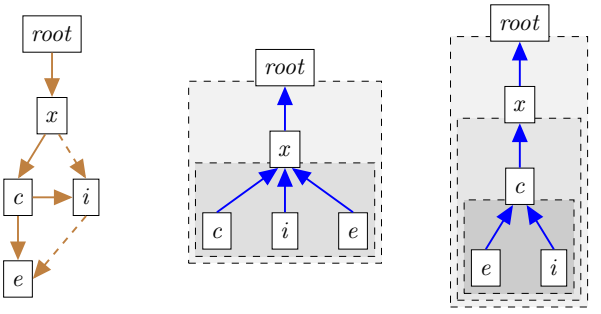
5.3 Comparing Universe Types vs. Ownership Types

In this section, we compare Universe Types, which enforce the owner-as-modifier encapsulation discipline, to Ownership Types, which enforce the owner-as-dominator encapsulation discipline, using examples we observed in the benchmarks.



(a) Object graph (b) OT tree (c) UT tree

Fig. 11. Write access to enclosing context results in flatter structure for UT as compared to OT (The bold edge from j to k highlights the write access)



(a) Object graph (b) OT tree (c) UT tree

Fig. 12. Readonly sharing of internal representation results in flatter structure for OT as compared to UT (The dotted edge from i to e highlights the readonly access)

In some cases, Universe Types inferred flatter structures than Ownership Types. This happens when an object j modifies an object k in an enclosing context. For example, consider Fig. 11. If object j *modifies* k, j and k must be peers in UT, which will force the flat ownership tree in Fig. 11(c). In contrast, OT reflects dominance and produces the deeper ownership tree shown in Fig. 11(b).

In other cases, Ownership Types inferred flatter structures than Universe Types. OT disallows exposure of internal objects outside of the boundary of the owner. UT is more permissive, in the sense that it allows readonly exposure. Consider Fig. 12. which represents a container c, its internal representation e and an iterator i over e. The OT tree is flatter because the iterator i creates a path to e which does not go through c. Therefore, c, e, and i must have x as their owner. In contrast, UT allows the exposure of i to x because this exposure is readonly. Therefore, c remains the owner of both e and i.

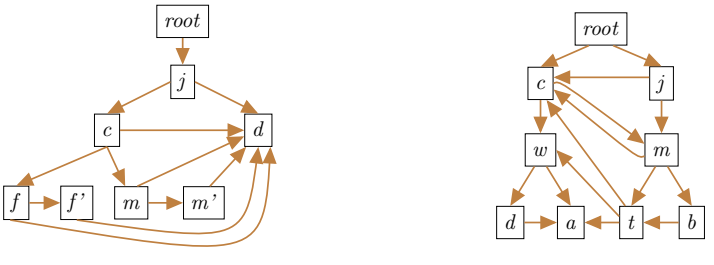
Fig. 13 compares OT and UT on the benchmarks. We consider only allocation sites, excluding strings and boxed primitives. Allocation sites provide the best approximation of ownership structure. On average 25% of the OT $\langle \mathsf{rep}|\_\rangle$ sites are typed rep in UT as well. On the other hand, on average 64% of the UT

Benchmark OT: / UT:	⟨rep|_⟩ / rep	⟨rep|_⟩ / peer	not ⟨rep|_⟩ / rep	not ⟨rep|_⟩ / not rep
JOlden	26 (22%)	8 (7%)	19 (16%)	66 (55%)
tinySQL	32 (6%)	123 (24%)	13 (2%)	355 (68%)
htmlparser	27 (2%)	234 (20%)	16 (1%)	926 (77%)
ejc	44 (2%)	336 (12%)	81 (3%)	2321 (83%)
javad	6 (10%)	38 (66%)	0 (0%)	14 (24%)
SPECjbb	75 (26%)	84 (29%)	25 (9%)	110 (37%)
jdepend	13 (7%)	71 (41%)	1 (1%)	90 (51%)
classycle	1 (0%)	109 (45%)	5 (2%)	128 (53%)

Fig. 13. Ownership Types vs. Universe Types on allocation sites. The four columns give the number of OT/UT pairings and, in parenthesis, the corresponding percentages. For example, column ⟨rep|_⟩/peer shows the number of allocation sites that were inferred as rep in OT and peer in UT.

rep sites are typed ⟨rep|_⟩ in OT as well. The discrepancy shows that it may be more common to have write access to enclosing context (which lowers rep to peer in UT), than it is to have readonly sharing of internal structure (which allows an object to stay rep in UT while it is not rep in OT). On average 40% of all allocation sites are inferred as rep in OT, and 14% are inferred as rep in UT, which suggests that write access to enclosing context is more common than readonly sharing of internal structure. The results suggest that in general, UT and OT capture distinct ownership structure. Note that as expected, there is a significantly larger percentage of rep allocation sites in UT compared to rep variables.

To further understand the differences between UT and OT, we examined the results of two of the benchmarks, javad and SPECjbb. Fig. 14(a) shows a partial object graph for javad. Here j represents the jvmDump object, c is the classFile object, f and f' are the fieldSection and fieldInfo objects, and m and m' are the methodSection and methodInfo objects. d is the DataStream object. All of c, f, f', m and m' *modify* d, which is an object from enclosing context. This forces all c, f, f', m, m' and d to be peers, and children of j in the UT ownership tree. Edges $c \to f, f \to f', c \to m$ and $m \to m'$ are ⟨rep|_⟩ in OT, but are peer in UT.



(a) javad (b) SPECjbb

Fig. 14. Partial object graphs for the javad and SPECjbb case studies

Now consider Fig. 14(b). It shows a partial object graph for SPECjbb. c is the Company object, w is a Warehouse object, d is a District object, and a is Address object which represents the District's address. j is a JBBmain thread, m is a TransactionManager, t is a Transaction and b is an array that stores transactions. Edges $w \rightarrow a$ and $t \rightarrow a$ expose the Address object outside of its creating object d. Therefore, the edge $d \rightarrow a$ cannot be $\langle \mathsf{rep}|\_\rangle$ in OT. However, the exposure is readonly, and it remains rep in UT.

6 Related Work

We discuss related work on ownership inference as well as other work on inference of pluggable and extended types.

Several dynamic approaches for ownership inference exist [9,18,21,27]. Although a dynamic approach may produce more precise results, it is inherently unsound and incurs a significant performance overhead. Also, it is difficult to generalize a dynamic approach to different type systems. In contrast, our approach is static and can be applied to multiple type systems.

Aldrich et al. [1] present an ownership type system and a type inference algorithm. Their inference creates equality, component and instantiation constraints and solves these constraints. Our inference solves different kinds of constraints, namely subtyping and adapt constraints.

Ma and Foster [16] propose Uno, a static analysis for automatically inferring ownership, uniqueness, and other aliasing and encapsulation properties in Java. Uno infers "stricter" ownership in which an owned object can only be accessed by its owner. Our inference has a less-restrictive ownership model. Uno's inference is based on Soot and it is difficult to map the inference results back to the source code, subsequently inhibiting type checking. Our type inference is integrated into the Checker Framework; we perform type checking as well.

Greenfieldboyce and Foster [11] present a framework called JQual for inferring user-defined type qualifiers in Java. JQual is effective for *source-sink* type systems, for which programmers need to add annotations to the sources and sinks and JQual infers the intermediate annotations for the rest of the program. Our tool handles more complex type systems such as Ownership type systems. In addition, JQual does not scale well in its field-sensitive mode as reported by Artzi et al.[2]. In contrast, our inference scales to programs of up to 110kLOC.

Chin et al. [3] propose CLARITY for the inference of user-defined qualifiers for C programs based on user-defined rules, which can also be inferred given user-defined invariants. CLARITY infers several type qualifiers, including pos and neg for integers, nonnull for pointers, and tainted and untainted for strings. These type qualifiers are not context-sensitive. Our tool focuses on type systems for Java, and it is context-sensitive (viewpoint adaptation models context sensitivity).

Dietl et al. [7] present a tunable static inference for Generic Universe Types (GUT). Constraints of GUT are encoded as a boolean satisfiability problem, which is solved by a weighted Max-SAT solver. The inference is tunable in the

sense that programmers can direct the inference by setting different weights or partially annotating the source code. In contrast, our inference can only be tuned by accepting programmers' manual annotations. However, by defining a ranking over typings, we avoid the exponential SAT solver and manage to scale to larger programs. A detailed comparison is left as future work.

Milanova and Vitek [17] present a static dominance inference analysis, based on which they perform Ownership Type inference. Our current work is an improvement over [17]. First, it accepts manual annotations to direct the inference, while [17] does not. Second, it provides optimality guarantees, while the inference in [17] does not provide guarantees — in theory, it may end up with a solution which produces a flat ownership tree. Third, our work includes a type checker which is not available in [17], and it works on more and larger benchmarks.

Sergey and Clark [23] introduce the notion of *gradual ownership types* and a corresponding consistent-subtyping relation. Their formalism provides a static guarantee of ownership invariants for fully annotated programs, but requires dynamic checks for partially-annotated programs. Their prototype works on non-generic Java programs and they analyzed 8,200 lines of code. In contrast, our inference is static and works on Java programs of up to 110kLOC.

Work on introducing generics to Java [10,15] solves similar challenges, because leaving every type as raw is a legal typing, but a useless one that expresses no design intent and detects no coding errors. In contrast to our work, Donovan et al. [10] use heuristics to find desirable solutions and their inference requires a pointer analysis. Kieżun et al. [15] make use of type constraints to ensure behavior preservation. They also use heuristics, otherwise user's input is required.

Our algorithm for computing the set-based solution (Sect. 4.1) is similar to the algorithm used by Tip et al. [15,24]. Both algorithms start with sets containing all possible answers and iteratively remove elements that are inconsistent with the typing rules. Our work differs as we introduce a ranking over valid typings and use the ranking to guide the automatic inference towards a final "best" typing.

Our work, as well as [24], falls in the category of type-based and constraint-based analysis, originally proposed by Palsberg and Schwartzbach [24].

7 Conclusion

We presented a unified framework for type inference and type checking of ownership type systems, and instantiated the framework for two such systems: Universe Types and Ownership Types. We presented a heuristic ranking over valid typings, and an efficient inference approach that produced maximal typings. We implemented the approach on top of the Checker Framework and presented results for Universe Types and Ownership Types on benchmarks of up to 110kLOC.

Acknowledgments. We thank the anonymous reviewers for their extensive feedback. This work was supported by NSF grants CNS-0855252 and CCF-0642911.

References

1. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, pp. 311–330 (2002)
2. Artzi, S., Kieżun, A., Quinonez, J., Ernst, M.D.: Parameter reference immutability: formal definition, inference tool, and comparison. ASE 16, 145–192 (2008)
3. Chin, B., Markstrum, S., Millstein, T., Palsberg, J.: Inference of User-Defined Type Qualifiers and Qualifier Rules. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 264–278. Springer, Heidelberg (2006)
4. Clarke, D., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA, vol. 33, pp. 48–64 (1998)
5. Cunningham, D., Dietl, W., Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: Universe Types for Topology and Encapsulation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 72–112. Springer, Heidelberg (2008)
6. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K., Schiller, T.W.: Building and Using Pluggable Type-Checkers. In: ICSE, pp. 681–690 (2011)
7. Dietl, W., Ernst, M.D., Müller, P.: Tunable Static Inference for Generic Universe Types. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 333–357. Springer, Heidelberg (2011)
8. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology 4(8), 5–32 (2005)
9. Dietl, W., Müller, P.: Runtime Universe type inference. In: IWACO (2007)
10. Donovan, A., Kieżun, A., Tschantz, M.S., Ernst, M.D.: Converting Java programs to use generic libraries. In: OOPSLA, pp. 15–34 (2004)
11. Greenfieldboyce, D., Foster, J.S.: Type qualifier inference for Java. In: OOPSLA, pp. 321–336 (2007)
12. Huang, W., Milanova, A.: Towards effective inference and checking of ownership types. In: IWACO (2011)
13. Huang, W., Milanova, A.: A Type System for Reference Immutability. Technical report, Rensselaer Polytechnic Institute, Department of Computer Science (2011)
14. Huang, W., Milanova, A.: On optimality of ownership type inference. Poster at ECOOP (2011)
15. Kieżun, A., Ernst, M.D., Tip, F., Fuhrer, R.M.: Refactoring for parameterizing Java classes. In: ICSE, pp. 437–446 (2007)
16. Ma, K.-K., Foster, J.S.: Inferring aliasing and encapsulation properties for Java. In: OOPSLA, pp. 423–440 (2007)
17. Milanova, A., Vitek, J.: Static Dominance Inference. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 211–227. Springer, Heidelberg (2011)
18. Mitchell, N.: The Runtime Structure of Object Ownership. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 74–98. Springer, Heidelberg (2006)
19. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer-Verlag New York, Inc. (1999)
20. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: ISSTA, pp. 201–212 (2008)
21. Potanin, A., Noble, J., Biddle, R.: Checking ownership and confinement. Concurrency and Computation: Practice and Experience 16(7), 671–687 (2004)
22. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In: PLDI, pp. 164–174 (2011)

23. Sergey, I., Clarke, D.: Gradual Ownership Types. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 579–599. Springer, Heidelberg (2012)
24. Palsberg, J., Schwartzbach, M.I.: Object-Oriented Type Systems. John Wiley and Sons (1994)
25. Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: OOPSLA, pp. 211–230 (2005)
26. Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J.: A Type System for Data-Centric Synchronization. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 304–328. Springer, Heidelberg (2010)
27. Vetchev, M., Yahav, E., Yorsh, G.: PHALANX: Parallel Checking of Expressive Heap Assertions. In: ISMM, pp. 41–50 (2010)

Object Initialization in X10

Yoav Zibin[1], David Cunningham[2], Igor Peshansky[1], and Vijay Saraswat[2]

[1] Google (work done at IBM)
{yzibin,igorp}@google.com
[2] IBM research in TJ Watson
{dcunnin,vsaraswa}@us.ibm.com

Abstract. X10 is an object oriented programming language with a sophisticated type system (constraints, class invariants, non-erased generics, closures) and concurrency constructs (asynchronous activities, multiple places). Object initialization is a cross-cutting concern that interacts with all of these features in delicate ways that may cause type, runtime, and security errors. This paper discusses possible designs for object initialization, and the "hardhat" design chosen and implemented in X10 version 2.2. Our implementation includes a fixed-point interprocedural (intra-class) data-flow analysis that infers, for each method called during initialization, the set of fields that are read, and those that are asynchronously and synchronously assigned. Our codebase of more than 200K lines of code only had 104 annotations. Finally, we formalize the essence of initialization checking with an effect system intended to complement a standard FJ style formalization of the type system for X10. This system is substantially simpler than the masked types of [10], and it is more practical (for X10) than the free-committed types of [12]. This is the first formalization of a type and (flow-sensitive) effect system for safe initialization in the presence of concurrency constructs.

1 Introduction

Constructing an object in a safe way is not easy: it is well known that dynamic dispatch or leaking `this` during object construction is error-prone [2,11,6], and various type systems and verifiers have been proposed to handle safe object initialization [7,14,4,10]. As languages become more and more complex, new pitfalls are created due to the interactions among language features.

X10 is an object oriented programming language with a sophisticated type system (constraints, class invariants, non-erased generics, closures) and concurrency constructs (asynchronous activities, multiple places). This paper shows that object initialization is a cross-cutting concern that interacts with other features in the language. We discuss several language designs that restrict these interactions, and explain why we chose the *hardhat* design for X10.

Hardhat [6] is a design that prohibits dynamic dispatch or leaking `this` (e.g., storing `this` in the heap) during construction. Such a design limits the user but also protects her from future bugs (see Fig. 1 below for two such bugs). X10's hardhat design is more complex due to additional language features such as concurrency, places, and closures.

On the other end of the spectrum, Java and C# allow dynamic dispatch and leaking `this`. However, they still maintain type and runtime safety by relying on the fact that every type has a default value (also called zero value, which is either 0, `false`, or `null`), and all fields are zero-initialized before the constructor begins. As a consequence, a

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 207–231, 2012.

half-baked object can leak before all its fields are set. Phrased differently, when reading a final field, one can read the default value initially and later read a different value. Another source of subtle bugs is related to the synchronization barrier at the end of a constructor [9] after which all assignments to final fields are guaranteed to be written. The programmer is warned (in the documentation only!) that immutable objects (using final fields) are thread-safe only if `this` does not escape its constructor. Finally, if the type-system is augmented, for example, with non-null types, then a default value no longer exists, which leads to complicated type-systems for initialization [4,10].

C++ sacrifices type-safety on the altar of performance: fields are not zero-initialized. (X10 has both type-safety and the performance for not zero-initializing fields.) Therefore if `this` leaks in C++, one can read an uninitialized field resulting in an arbitrary value. Moreover, method calls are statically bound during construction, which may result in an exception at runtime if one tries to invoke a virtual method of an abstract class (see Fig. 3b below). (Determining whether this happens is intractable [5].) We believe a design for object initialization should have these desirable properties:

Cannot read uninitialized fields. One should not be able to read uninitialized fields. In C++ it is possible to read uninitialized fields, returning an unspecified value which can lead to unpredictable behavior. In Java, fields are zero initialized before the constructor begins to execute, so it is possible to read the default or zero value, but never an unspecified value.

Single value for final fields. Final fields can be assigned exactly once, and should be read only after assigned. In Java it is possible to read a final field before it was assigned, therefore returning its default value.

Immutable objects are thread-safe. Immutable classes are a common pattern where fields are final/const and instances have no mutable state, e.g., `String` in Java. Immutable objects are often shared among threads without any explicit synchronization, because programmers assume that if another thread gets a handle to an object, then that thread should see all assignments done during initialization. However, weak memory models today do not necessarily have this guarantee and immutable objects could be thread-*unsafe*! Sec. 1.3 below will show that this can happen in Java if `this` escapes from the constructor [9].

Simple. The order of initialization should be clear from the syntax, and should not surprise the user. Dynamic dispatch during construction disrupts the order of initialization by executing a subclass's method before the superclass finished its initialization. This kind of initialization order is error-prone and often surprises the user.

Flexible. The user should be able to express the common idioms found in other languages with minor variations.

Type-safe. The language should continue to be statically type-safe even if it has rich types that do not have a default or zero value, such as non-null types (`T{self!=null}` in X10's syntax). Type-safety implies that reading from a non-null type should never return `null`. Adding non-null types to Java [3,4,10] has been a challenge precisely due to Java's relaxed initialization rules.

We took the ideas of prohibiting dynamic dispatch or leaking `this` during construction from [6], and materialized them into a set of rules that cover all aspects of X10 (type-system, closures, generics, properties, and concurrent and distributed constructs). This hardhat design in X10 (version 2.2) has the above desirable properties, however

they come at a cost of limiting flexibility: it is not possible to express cyclic immutable structures in X10. We chose simplicity over flexibility in our design choices, e.g., X10 prohibits creating an alias of `this` during object construction (whereas a more flexible design could track aliases via alias-analysis, at the cost of sacrificing simplicity). To our knowledge, X10 is the first object-oriented (OO) language to adopt the strict hardhat initialization design.

Because one cannot read uninitialized fields in X10, there is no need zero-initialize the object's fields (as done in Java before the constructor executes). A recent study [13] measured the direct cost of *zero initialization*, which "is surprisingly high: up to 12.7%, with average costs ranging from 2.7 to 4.5% on a high performance virtual machine on IA32 architectures." (Note that the indirect costs due to caching might even be higher.)

X10 version 2.0 till 2.2 had an alternative initialization design called *proto* that allowed cyclic immutable structures at the cost of a more complicated design. In OOPSLA'11, Summers and Müller [12] presented an initialization type-system that is almost identical to our proto proposal, but with different terminology: fully initialized objects are termed *committed* (non-proto), and objects under initialization are termed *free* (proto). Whereas in our proto proposal one cannot read uninitialized fields, in Summers' type-system reading uninitialized fields is allowed and it returns an *unclassified* type (and reading non-null fields from an unclassified type is allowed but it may return null). Phrased differently, reading a field is always allowed, but it may return null even for non-null fields. In contrast, X10 and C++ have types that cannot contain null (in C++ only pointers may be null, and X10 has *structs* which are inlinable objects that may not contain null), thus Summers' type-system is not applicable in such languages. Moreover, proto was used in X10 in the past 3 years prior to X10 2.2, and it was made obselete in favor of the hardhat design presented in this paper because proto did not work well in practice. For example, consider an implementation of `LinkedList` that has a non-null field `header`:

```
class LinkedList extends SuperClass {
  final Entry header = new Entry();
  LinkedList(Collection c) { addAll(c); }
  public boolean addAll(Collection c) {
    // this.header is null if SuperClass calls addAll in its constructor.
  }
}
```

During construction we must read from `this.header`, but `this` is still proto and it is illegal to read from a proto object (in Summer's type-system, reading is allowed but it may return null). It is tempting to think that a dataflow algorithm can prove that `header` was already assigned, however the dataflow must be inter-procedural, and it is further complicated by overriding and `this` escaping. For example, if `SuperClass` calls `addAll` in its constructor, then `header` will still have the default value of `null`. The newly proposed hardhat design can modularly type-check this code assuming that `addAll` is annotated as non-escaping (see Sec. 2).

The *contributions of this paper* are: (i) a complete and strict hardhat design in a full-blown advanced OO language with many cross-cutting concerns (default values, final fields, dataflow analysis, overriding, and especially the concurrent and distributed aspects), (ii) an inter-procedural fixed-point algorithm for definite-async assignment, (iii) implementation inside the X10 open-source compiler and converting the entire X10 code-base (+200K lines of code) to conform to the hardhat principles, (iv) FX10

formalism which is the first to present a flow-sensitive effect system with concurrency constructs and a soundness theorem stating that one can never read an uninitialized field in a statically correct program.

For object initialization rules, the details matter. Instead of basing our work on abstract theoretical discussions, we have chosen to work with a concrete language (X10, see Sec. 3) in which all these rules have been worked out to illustrate the subtleties involved. Our analysis and design will be applicable to any OO language with fine-grained concurrency. Object initialization rules must be dealt with in order to support determinate computation. For example, *Deterministic Parallel Java* (DPJ) [1] also have similar rules for object initialization to prevent `this` from leaking: "the DPJ type and effect system ensures that no other task can access `this` until after the constructor returns."

The remainder of this introduction presents common initialization pitfalls (in sequential, concurrent, and distributed code in both Java and X10) and how the hardhat design prevents them. Specifically, it presents initialization pitfalls in sequential code (Sec. 1.1), concurrent code (Sec. 1.3), and distributed X10 code (Sec. 1.4), and the crux of the hardhat design that prevents these sequential pitfalls (Sec. 1.2).

1.1 Initialization Pitfalls in Sequential Code

Fig. 1a demonstrates the two most common initialization pitfalls in Java: leaking `this` and dynamic dispatch. We will first explain the surprising output due to dynamic dispatch, and then the less known possible bug due to leaking `this`.

Executing `new B()` prints `a=42,b=0`, which is surprising to most Java users. One would expect `b` to be 2, and `a` to be either 1 or 44. However, due to initialization order and dynamic dispatch, the user sees the default value for `b` which is 0, and therefore the value of `a` is 42. We will trace the initialization order for `new B()`: we first allocate a new object with zero-initialized fields, and then invoke the constructor of `B`. The constructor of `B` first calls `super()`, and only afterward it will run the field initializer which sets `b` to 2. This is the cause of surprise, because *syntactically* the field initializer comes before `super()`, however it is executed after. (And writing `b=2;super();` is illegal in Java because calling super must be the first statement). During the `super()` call we perform two dynamic dispatches: the two calls (`initA()` and `toString()`) execute the implementation in `B` (and recall that `b` is still 0). Therefore, `initA()` returns 42, and `toString()` returns `a=42,b=0`. This bug might seem pretty harmless, however if we change the type of `b` from `int` to `Integer`, then this code will throw a `NullPointerException`, which is more severe.

The second pitfall is leaking `this` before the object is fully-initialized, for example, `S.add(this)`. Note that we leak a partially-initialized object, i.e., the fields of `B` have not yet been assigned and they contain their default values. Suppose that some other thread iterates over `S` and prints them. Then that thread might read `b=0`. In fact, it might even read `a=0`, even though we just assigned 42 to `a` two statements ago! The reason is that this write is guaranteed to be seen by other threads only after an implicit synchronization barrier that is executed after the constructor ends. Sec. 1.3 further explains final fields in Java and the implicit synchronization barrier.

1.2 The Crux of the Hardhat Design

The hardhat design in X10 (described in Sec. 2) prevents both pitfalls, because its rules allow dynamic dispatching only when `this` cannot be accessed (first pitfall) and prohibit

```
class A {
 static HashSet S = new HashSet();
 final int a;
 A() {
  a = initA(); // dynamic dispatch!
  System.out.println(toString());
  S.add(this); // leakage!
 }
 int initA() { return 1; }
 public String toString() {
  return "a="+a; }
}
class B extends A {
 int b = 2;
 int initA() { return b+42; }
 public String toString() {
  return super.toString()+",b="+b;}
}
```

(a) Initialization pitfalls

```
class A {
 static HashSet S = new HashSet();
 final int a;
 protected A() {
  a = initA(); // ok
  System.out.println(toStringOfA());
  // S.add(this); // Would be an error
 }
 @NoThisAccess int initA() {return 1;}
 public String toString() {
  return toStringOfA(); }
 @NonEscaping final String toStringOfA(){
  return "a="+a; }
 public static A createA() {
  A res = new A(); S.add(res);
  return res;
 }
}
class B extends A {
 int b = 2;
 @NoThisAccess int initA() {return 42;}
 public String toString() {
  return super.toString()+",b="+b;}
 public static B createB() {
  B res = new B(); S.add(res);
  return res;
 }
}
```

(b) Fixed to conform to the hardhat design

Fig. 1. Two initialization pitfalls in Java: leaking `this` and dynamic dispatch

leaking `this` (second pitfall). We use two method annotations to mark that a method is non-escaping: `@NonEscaping` and `@NoThisAccess`; the first prohibits leaking `this`, and the second is even more strict and prohibits any access of `this`. The essence of the hardhat design are these two rules: (i) Constructors and non-escaping methods may leak/alias `this` *only* to other non-escaping methods (i.e., `this` can only be used as the receiver of a non-escaping method call), (ii) Non-escaping methods are either private or final (thus they cannot be overridden), except `@NoThisAccess` methods that may be overridden but they cannot access `this`. These two rules prevent the two pitfalls of leaking `this` and dynamic dispatching.

Initialization in X10 has the following main attributes: (i) `this` is the only accessible raw/uninitialized object in scope, (ii) only `@NoThisAccess` methods can be dynamically dispatched during construction, (iii) one can read a field *only* after it was assigned, and all fields are assigned by the time the constructor finishes, (iv) reading a final field always results in the same value. (In contrast to Java and [12] where reading a final field might return different values at different times.) Furthermore, with the hardhat rules there is even no need to *zero-initialize all fields* before executing the constructor (as done in Java), thus reducing the program runtime. (We are now in the process of measuring this reduction in runtime; Using a simple bytecode verifier it is possible to ensure that this optimization is safe.)

Fig. 1b shows how to convert the code of Fig. 1a to the hardhat design and thus avoid these two pitfalls (but the original program behavior is changed). We made the following changes: (i) `toString` now delegates to a final non-escaping method `toStringOfA`, and the constructor of `A` can call `toStringOfA`; `B` cannot override this method because it is final, (ii) `initA` is `@NoThisAccess` and therefore `B.initA` cannot read the field `b` (which has not been assigned yet), (iii) instead of leaking `this` into `S` in the constructor of `A`, we refactored the code into two factory methods that create instances of `A` and `B`, and only then add the fully-initialized instance to `S`.

1.3 Initialization Pitfalls in Concurrent Code

We will start with an anecdote: suppose you have a friend that playfully removed all the occurrences of the `final` keyword from your legal Java program. Would your program still *run* the same? On the face of it, `final` is used only to make the *compiler* more strict, i.e., to catch more errors at compile time (to make sure a method is not overridden, a class is not extended, and a field or local is assigned exactly once). After *compilation* is done, `final` should not change the *runtime* behavior of the program. However, this is not the case due to interaction between initialization and concurrency: a synchronization barrier is implicitly added at the end of a constructor [9] ensuring that assignments to *final* fields are visible to all other threads. (Assignments to non-final fields might not be visible to other threads!)

The synchronization barrier was added to the memory model of Java 5 to ensure that the common pattern of immutable objects is thread-safe. The memory model does not guarantee *sequential consistency*, but only *weak consistency*. (The barrier would not be needed with sequential consistency.) Without this barrier another thread might see the default value of a field instead of its final value. For example, it is well-known that `String` is immutable in Java, and its implementation uses three final fields: `char[] value`, and two `int` fields named `offset` and `count`. The following code `"AB".substring(1)` will return a new string `"B"` that shares the same `value` array as `"AB"`, but with `offset` and `count` equal to 1. Without the barrier, another thread might see the default values for these three fields, i.e., `null` for `value` and 0 for `offset` and `count`. For instance, if one removes the `final` keyword from all three fields in `String`, then the following code might print `B` (the expected answer), or it might print `A` or an empty string, or might even throw a `NullPointerException`:

```
final String name = "AB".substring(1);
new Thread() { public void run() {
    System.out.println(name); } }.start();
```

A similar bug might happen in Fig. 1a because `this` was leaked into `S` before the barrier was executed. Consider another thread that iterates over `S` and reads field `a`. It might read 0, because the assignment of 42 to `a` is guaranteed to be visible to other threads only after the barrier was reached.

Java's documentation recommends using final fields when creating an immutable class, and avoid leaking `this` in the constructor. However, `javac` does not even give a warning if that recommendation is violated. To summarize, final fields in Java enable thread-safe immutable objects, but the user must be careful to avoid the pitfall of leaking `this`. The hardhat design in X10 prevents any leakage of `this`, thus making it safe and easy to create immutable classes.

1.4 Initialization Pitfalls in Distributed X10 Code

X10 supports parallelism in the form of both concurrent and distributed code. Next we describe parallelism in X10 and its interaction with object initialization.

Concurrent code uses asynchronous un-named activities that are created with the `async` construct, and it is possible to wait for activities to complete with the `finish` construct. Informally, statement `async S` executes statement `S` asynchronously; we say that the newly created activity *locally terminated* when it finished executing `S`, and that it *globally terminated* when it locally terminated and any activity spawned by `S` also globally terminated. Statement `finish S` blocks until all the activities created by `S` globally terminated.

Distributed code is run over multiple *places* that do *not* share memory, therefore objects are (deeply) copied from one place to another. The expression `at(p) E` evaluates `p` to a place, then copies all captured references in `E` to place `p`, then evaluates `E` in place `p`, and finally copies the result back to the original place. Note that `at` is a synchronous construct, meaning that the current activity is blocked until the `at` finishes. This construct can also be used as a statement, in which case there is no copy back (but there is still a notification that is sent back when the `at` finishes, in order to release the blocked activity in the original place).

Fig. 2a (and Fig. 2b) shows how to (correctly) calculate the Fibonacci number `fib(n)` in X10 using concurrent and distributed code. The keywords `val` and `var` are modifiers that correspond to final and non-final variables, respectively. Note how `fib(n-2)` is calculated asynchronously at the next place (`next()` returns the next place in a cyclic ordering of all places), while simultaneously recursively calculating `fib(n-1)` in the current place (that will recursively spawn a new activity, and so on). Therefore, the computation will recursively continue to spawn activities at the next place until `n` is 1. When both calculations globally terminate, the `finish` unblocks, and we sum their result into the *final* field `fib`.

We note that using *final local variables* for `fib2` and `fib1` instead of fields would have made this example more elegant, however we chose the latter because this paper focuses on *object* initialization. X10 has similar initialization rules for final locals and final fields, but it is outside the scope of this paper to present all forms of initialization in X10 (including local variables and static fields). Details of those can be found in X10's language specification at `x10-lang.org`.

There are two possible pitfalls in this example. The first is a distributed pitfall, where one assigns to a field of a copy of `this` in another place (instead of assigning in the original place). Leaking `this` to another place before it is fully initialized might also cause bugs in custom serialization code (see Sec. 2.10). The second is a concurrency pitfall, where we forget to use `finish`, and therefore we might read from a field before its assignment was definitely executed. Java has definite-assignment rules (using an intra-procedural data-flow analysis) to ensure that a read can only happen after a write; The hardhat design in X10 adopted those rules and extended them in the face of concurrency to support the pattern of *asynchronous initialization* where an `async` must have an enclosing `finish` (using an intra-class inter-procedural analysis, see Sec. 2.11).

The hardhat design in X10 prevents both pitfalls by ensuring that all fields of an object are definitely-synchronously assigned when construction of that object ends, and that only fully initialized objects can cross places.

```
class Fib {
 val fib2:Int, fib1:Int, fib:Int;
 def this(n:Int) {

  async {
   val p = here.next();
   at(p) if (n<=1)
    fib2 = 0; else // Err1
    fib2 = new Fib(n-2).fib; // Err1
  }

  if (n<=0)
   fib1 = 0;
  else if (n<=1)
   fib1 = 1;
  else
   fib1 = new Fib(n-1).fib;
  fib = fib2+fib1; // Err2
 }
}
```

(a) Initialization pitfalls in X10

```
class Fib {
 val fib2:Int, fib1:Int, fib:Int;
 def this(n:Int) {
  finish {
   async {
    val p = here.next();
    fib2 = at(p) (n<=1) ?
     0 :
     new Fib(n-2).fib;
   }
  }
  if (n<=0)
   fib1 = 0;
  else if (n<=1)
   fib1 = 1;
  else
   fib1 = new Fib(n-1).fib;
  fib = fib2+fib1;
 }
}
```

(b) Fixed to conform to the hardhat design

Fig. 2. Concurrent and distributed Fibonacci example in X10. Concurrent code is expressed using `async` and `finish`: `async` starts an asynchronous activity, and `finish` waits for all spawned activities to finish. Distributed code uses `at` to shift among *places*; `here` denotes the current place. `at(p) E` evaluates expression `E` in place `p`, and finally copies the result back; any final variables captured in `E` from the outer environment (e.g., `n`) are first copied to place `p`. The two initialization pitfalls: (1) write to field `this.fib2` in another place, which causes (an uninitialized) `this` to be copied to `p`, so one writes to a *copy* of `this` (and the original object is never fully initialized!), (2) read from `fib2` before its write definitely *finished*.

The rest of this paper is organized as follows. Sec. 2 presents the hardhat initialization rules of X10 version 2.2 using examples, by slowly adding language features and describing their interaction with object initialization. Sec. 3 outlines our implementation within the X10 compiler using the polyglot framework, the compilation time overhead of checking these initialization rules, and the annotation overhead in our X10 code base. Sec. 4 presents Featherweight X10 (FX10), which is a formalization of core X10 that includes `finish`, `async`, and flow-sensitive type-checking rules. Sec. 5 summarizes previous work in the field of object initialization. Finally, Sec. 6 concludes.

2 X10 Initialization Rules

X10 is an advanced object-oriented language with a complex type-system and concurrency constructs. This section describes how object initialization interacts with X10 features. We begin with object-oriented features found in mainstream languages, such as constructors, inheritance, dynamic dispatch, exceptions, and inner classes. We then proceed to X10's type-system features, such as constraints, properties, class invariants, closures, (non-erased) generics, and structs, followed by the parallel features of X10 for

writing concurrent code (`finish` and `async`), and distributed code (`at`). Finally, we describe the inter-procedural data-flow analysis that ensures that a field is read only after it has been assigned.

2.1 Constructors and Inheritance

Inheritance is the first feature that interacts with initialization: when class `B` inherits from `A` then every instance of `B` has a sub-object that is like an instance of `A`. When we initialize an instance of `B`, we must first initialize its `A` sub-object. We do this in X10 by forcing the constructors of `B` to make a super call, i.e., call a constructor of `A` (either explicitly or implicitly).

Fig. 3 shows X10 code that demonstrates the interaction between inheritance and initialization, and explains by example why leaking `this` during construction can cause bugs. In all the examples, all errors issued by the X10 compiler are marked with `//err` (and if there is no such mark then the code is correct).

We say that an object is *raw* (also called partially initialized) before its constructor ends, and afterward it is *cooked* (also called fully initialized). Note that when an object is cooked, all its sub-objects must be cooked as well. X10 prohibits any aliasing or leaking of `this` during construction, therefore only `this` or `super` can be raw (any other variable is definitely cooked).

Object initialization begins by invoking a constructor, denoted by the method definition `def this()`. The first leak would cause a problem because field `a` was not assigned yet. However, even after all the fields of `A` have been assigned, leaking is still a problem because fields in a subclass (field `b`) have not yet been initialized. Note that leaking is not a problem if `this` is not raw, e.g., in `m1()`.

We begin with two definitions: (i) when an object is *raw*, and (ii) when a method is *non-escaping*. (i) Variables `this` and `super` are raw during the object's construction, i.e., in field initializers and in non-escaping methods (methods that cannot escape or leak `this`). (ii) Obviously constructors are non-escaping, but you can also annotate methods *explicitly* as `@NonEscaping`, or they can be inferred to be *implicitly* non-escaping if they are called on a raw `this` receiver.

For example, `m2` is *implicitly* non-escaping (and therefore cannot leak `this`) because of the call to `m2` in the constructor. The user could also mark `m2` *explicitly* as non-escaping by using the annotation `@NonEscaping`. (Like in Java, `@` is used for annotations in X10.) We recommend explicitly marking non-escaping methods as `@NonEscaping` to show intent, as done on method `m3`. Without this annotation the call `super.m3()` in `B` would be illegal, due to rule 2. (We could infer that `m3` must be non-escaping, but that would cause a dependency from a subclass to a superclass, which is not natural for people used to separate compilation.) Finally, we note that all errors in this example are due to rule 1 that prevents leaking a raw `this` or `super`.

2.2 Dynamic Dispatch

Dynamic dispatch may transfer control to the subclass before the superclass completed its initialization. Fig. 3b demonstrates why dynamic dispatch is error-prone during construction: calling `m1` in `A` would dynamically dispatch to the implementation in `B` that would read the default value.

```
class A {
 val a:Int;
 def this() {
  LeakIt.foo(this); //err
  this.a = 1;
  val me = this; //err
  LeakIt.foo(me);
  // so m2 is implicitly non-escaping
  this.m2();
 }
 // permitted to escape
 final def m1() {
  LeakIt.foo(this);
 }
 // implicitly non-escaping
 final def m2() {
  LeakIt.foo(this); //err
 }
 // explicitly non-escaping
 @NonEscaping final def m3() {
  LeakIt.foo(this); //err
 } }
class B extends A {
 val b:Int;
 def this() {
  super(); this.b = 2; super.m3();
 }
}
```

(a) Escaping `this` example

```
abstract class C {
 val a1:Int, a2:Int;
 def this() {
  // Can only call non-escaping methods
  this.a1 = m1(); //err1
  this.a2 = m2();
  m4(); m5();
 }
 abstract def m1():Int;
 @NoThisAccess abstract def m2():Int;
 @NonEscaping def m3():void {} // err
 @NonEscaping final def m4():void {}
 @NonEscaping private def m5():void {}
}
class D extends C {
 var b:Int = 3; // non-final field
 def m1() {
  val x = super.a1;
  val y = this.b;
  return 1;
 }
 @NoThisAccess def m2() {
  // Cannot use this or super
  val x = super.a1; //err2
  val y = this.b; //err3
  return 2;
 }
}
```

(b) Dynamic dispatch example

Fig. 3. Definition of *raw*: `this` and `super` are *raw* in non-escaping methods and in field initializers. **Definition of *non-escaping*:** A method is *non-escaping* if it is a constructor, or annotated with `@NonEscaping` or `@NoThisAccess`, or a method that is called on a raw `this` receiver. **Rule 1:** A raw `this` or `super` cannot escape or be aliased. **Rule 2:** A call on a raw `super` is allowed only for a `@NonEscaping` method. **Rule 3:** A non-escaping method must be private or final, unless it has `@NoThisAccess`. **Rule 4:** A method with `@NoThisAccess` cannot access `this` or `super` (neither read nor write its fields).

X10 prevents dynamic dispatch by requiring that non-escaping methods must be private or final (so overriding is impossible). For example, *`err1`* is caused by rule 3 because `m1` is neither private nor final nor `@NoThisAccess`.

However, sometimes dynamic dispatch is required during construction. For example, if a subclass needs to refine initialization of the superclass's fields. Such refinement cannot have any access to `this`, and therefore such methods must be marked with `@NoThisAccess`. For example, *`err2`* and *`err3`* are caused by rule 4 that prohibits access `this` or `super` when using `@NoThisAccess`. `@NoThisAccess` prohibits any access to `this`, however, one could still access the method parameters. (If the subclass needs to read a certain field of the superclass that was previously assigned, then that field can be passed as an argument.)

In C++, the call to `m1` is legal, but at runtime methods are statically bound, so you will get a crash trying to call a pure virtual function. In Java, the call to `m1` is also legal, but at runtime methods are dynamically bound, so the implementation of `m1` in `B` will read the default values of `a1` and `b`.

2.3 Exceptions

Constructing an object may not always end normally, e.g., building a date object from an illegal date string should throw an exception. Exceptions combined with inheritance interact with initialization in the following way: a cooked object must have cooked sub-objects, therefore if a constructor ends normally (thus returning a cooked object) then all preceding constructor calls (either `super(...)` or `this(...)`) must end normally as well. Phrased differently, in a constructor it should not be possible to recover from an exception thrown by a this or super constructor call. This is one of the reasons why a constructor call must be the first statement in Java; failure to verify this led to a famous security attack [2].

```
class B extends A {
  def this() {
    try { super(); } catch(e:Throwable){} //err
  }
}
```

Fig. 4. Exceptions example: if a constructor ends normally (without throwing an exception), then all preceding constructor calls ended normally as well. **Rule 5:** If a constructor does not call `super(...)` or `this(...)`, then an implicit `super()` is added at the beginning of the constructor; the first statement in a constructor is a constructor call (either `super(...)` or `this(...)`); a constructor call may only appear as the first statement in a constructor .

Fig. 4 shows that it is an error to try to recover from an exception thrown by a constructor call; the reason for the error is rule 5 that requires the first statement to be `super()`.

2.4 Inner Classes

Inner classes usually read the outer instance's fields during construction, e.g., a list iterator would read the list's header node. Therefore, X10 requires that the outer instance is cooked, and prohibits creating an inner instance when the receiver is a raw `this`.

Fig. 5a shows it is an error in X10 to create an inner instance if the outer is raw (from rule 6), but it is ok to create an instance of a static nested class, because it has no outer instance.

In fact, it is possible to view this rule as a special case to the rule that prohibits leaking a raw `this` (because when you create an inner instance on a raw `this` receiver, you create an alias of `this`, and now you have two raw objects: `Inner.this` and `Outer.this`). We wish to keep the invariant that only one `this` can be raw.

In our rules, we assume that there is a single `this` reference, because we can convert all inner, anonymous and local classes into static nested classes by passing the outer instance and all other captured variables explicitly as arguments to the constructor.

```
class Outer {
  val a:Int;
  def this() {
    // Outer.this is raw
    Outer.this. new Inner(); //err
    new Nested(); // ok
    a = 3;
  }
  class Inner {
    def this() {
      // Inner.this is raw, but
      // Outer.this is cooked
      val x = Outer.this.a;
    }
  }
  static class Nested {}
}
```

(a) Inner class example: the outer instance is always cooked.

```
class DefaultValuesExample {
  val i0:Int; //err
  // Note the fields below are non-final
  var i1:Int; //ok, has default
  // no default
  var i2:Int{self!=0}; //err
  // ok, has initializer
  var i3:Int{self!=0} = 3;

  var i4:Int{self==42}; //err

  var s1:String;
  var s2:String{self!=null}; //err

  var b1:Boolean;
  var b2:Boolean{self==true}; //err
}
```

(b) Default value example.

Fig. 5. Rule 6: a raw `this` cannot be the receiver of `new`.
Definition of *has-zero*: A type *has-zero* if it contains the zero value (which is either `null`, `false`, 0, or zero in all fields for user-defined structs) or if it is a type parameter guarded with `haszero` (see Sec. 2.8). **Rule 7:** A `var` field that lacks a field initializer and whose type has-zero, is implicitly given a zero initializer..

We now turn our attention to X10's sophisticated type-system features not found in mainstream languages: constraints, properties, class invariants, closures, (non-erased) generics, and structs.

2.5 Constraints and Default/Zero Values

X10 supports constrained types using the syntax `T{c}`, where `c` is a boolean expression that can use final variables in scope, literals, properties (described below), the special keyword `self` that denotes the type itself, field access, equality (`==`) and disequality (`!=`). There are plans to add arithmetic inequality (`<`, `<=`) to X10 in the future, and one can plug in any constraint solver into the X10 compiler.

As a consequence of constrained types, some types do not have a default value, e.g., `Int{self!=0}`. Therefore, in X10, the fields of an object cannot be zero-initialized as done in Java. Furthermore, in Java, a non-final field does not have to be assigned in a constructor because it is assumed to have an implicit zero initializer. X10 follows the same principle, and a non-final field is implicitly given a zero initializer *if its type has-zero*. Fig. 5b defines when a type *has-zero*, and gives examples of types without zero. Note that `i0` has to be assigned because it is a final field (`val`), as opposed to `i1` which is non-final (`var`).

2.6 Properties and the Class Invariant

Properties are final fields that can be used in constraints, e.g., `Array` has a `size` property, so an array of size 2 can be expressed as: `Array{self.size==2}`. The differences

between a property and a final field are both syntactic and semantic, as seen in class E of Fig. 6. Syntactically, properties are defined after the class name, must have a type and cannot have an initializer, and must be initialized in a constructor using a property call statement written as property(...). Semantically, properties are initialized before all other fields, and they can be used in constraints with the prefix self.

```
class E(a:Int) {
  def this(x:Int) {
       property(x);
  }
}
```

```
class F(b:Int) {b==a} extends E {
  val f1 = a+b, f2:Int, f3:E{this.a==self.a};
  def this(x:Int) {
   super(x);
   val i1 = super.a;
   val i2 = this.b; //err
   val i3 = this.f1; //err
   f2 = 2; //err (must be after property(x))
   property(x);
   f3 = new E(this.a);
  }
}
```

Fig. 6. Properties and class invariant example: properties (a and b) are final fields that are initialized before all other fields using a property call (property(...); statement). If a class does not define any properties, then an implicit property() is added after the (implicit or explicit) super(...). Field initializers are executed in their declaration order after the (implicit or explicit) property call. **Rule 8:** If a constructor does not call this(...), then it must have exactly one property call, and it must be unconditionally executed (unless the constructor throws an exception). **Rule 9:** The class invariant must be satisfied after the property call. **Rule 10:** The super fields can only be accessed after super(...), and the fields of this can only be accessed after property(...).

When using the prefix this, you can access both properties and other final fields. The difference between this and self is shown in field f3 in Fig. 6: this.a refers to the property a stored in this, whereas self.a refers to a stored in the object to which f3 refers. (In the constructor, we indeed see that we assign to f3 a new instance of E whose a property is equal to this.a.)

Properties must be initialized before other fields because field initializers and field types can refer to properties (see initializer for f1 and the type of f3). The superclass's fields can be accessed after the super call, and the other fields after the property call; field initializers are executed after the property call.

The *class invariant* ({b==a} in Fig. 6) may refer only to properties, and it must be satisfied after the property call (rule 9).

2.7 Closures

Closures are functions that can refer to final variables in the enclosing scope, e.g., they can refer to final method parameters, locals, and this. When a closure refers to a variable, we say that the variable is *captured* by the closure, and the variable is thus stored in the closure object. Closures interact with initialization when they capture this during construction.

Fig. 7a shows why it is prohibited to capture a raw this in a closure: that closure can later escape to another place which will serialize all captured variables (including the

```
class A {
 var a:Int = 3;
 def this() {
  val closure1 = ()=>this.a; //err
  at(here.next()) closure1();
  val local_a = this.a;
  val closure2 = ()=>local_a;
 }


}
```

(a) Closures example.

```
class B[T] {T haszero} {
 var f1:T;
 val f2 = Zero.get[T]();
}
struct WithZeroValue(x:Int,y:Int) {}
struct NoZeroValue(x:Int{self!=0}) {}
class Usage {
 var b1:B[Int];
 var b2:B[Int{self!=0}]; //err
 var b3:B[WithZeroValue];
 var b4:B[NoZeroValue]; //err
}
```

(b) `haszero` type predicate example.

Fig. 7. Rule 11: A closure cannot capture a raw `this`.
Rule 12: A type must be consistent, i.e., it cannot contradict method guards or class invariants.

raw `this`, which should not be serialized, see Sec. 2.10). The work-around for using a field in a closure is to define a local that will refer only to the field (which is definitely cooked) and capture the local instead of the field as done in `closure2`.

2.8 Generics and Structs

Structs in X10 are header-less inlinable objects that cannot inherit code (i.e., they can *implement* interfaces, but cannot *extend* anything). Therefore an instance of a struct type has a known size and can be inlined in a containing object. Java's primitive types (`int`, `byte`, etc) are represented as structs in X10. Structs, as opposed to classes, do not contain the value `null`.

Generics in X10 are reified, i.e, not erased as in Java. For example, a `Box[T]` has a single field of type `T`, and instances of `Box[Byte]` and `Box[Double]` have the same size in Java but different sizes in X10. Although generics are not a new concept, the combination of generics and the lack of default values leads to new pitfalls. For example, in Java and C#, it is possible to define an equivalent to

```
class A[T] { var a:T; }
```

However, this is illegal in X10 because we cannot be sure that `T` has-zero (see Fig. 5b), e.g., if the user instantiates `A[Int{self!=0}]` then field `a` cannot be assigned a zero value without violating type-safety. Therefore X10 has a type predicate written `X haszero` that evaluates to true if type `X` has-zero. Using `haszero` in a constraint (e.g., in a class invariant or a method guard), makes it possible to guarantee that a type-parameter will be instantiated with a type that has-zero.

Fig. 7b shows an example of a generic class `B[T]` that constrains the type-parameter `T` to always have a zero value. According to rule 7, field `f1` has an implicit zero field initializer. It is also possible to write the initializer explicitly (as done in field `f2`) by using the static method `Zero.get[X]()` (that is guarded by `X haszero`). Next we see two struct definitions: the first has two properties that has-zero, and the second has a property that does not have zero. According to the definition of has-zero in Fig. 5b, a struct has-zero if all its fields has-zero, therefore `WithZeroValue haszero` is true, but `NoZeroValue haszero` is false. Finally, we see an example of usages of `B[T]`, where two usages are legal and two are illegal (see rule 12).

We now turn our attention to the parallel features of X10 for concurrent programming (`finish` and `async`) and distributed programming (`at`). Sec. 1.4 already explained how parallel code is written in X10, and what are the common pitfalls of initialization in parallel code. Next we present the rules that prevent these pitfalls.

2.9 Concurrent Programming and Initialization

```
class A {
 var f1:Int; // note: var field
 val f2:Int; // note: val field
 val f3:Int;
 //err: f2 was not definitely assigned
 def this() {
  async f1 = 1; async f2 = 2;
  finish { async f3 = 3; }
 }
}
```

(a) Concurrency in initialization example: asynchronously assign to a field.

```
class A {
 val f:Int;

 //err: f was not definitely assigned
 def this() {
  // Execute at another place
  at (here.next())
   this.f = 1; //err: this escaped
 }
}
```

(b) Distributed initialization example.

Fig. 8. Rule 13: A constructor must finish assigning to all fields at least once. **Rule 14:** A final field can be assigned at most once.
Rule 15: a raw `this` cannot be captured by an `at`.

Fig. 8a shows how to asynchronously assign to fields. Recall that we wish to guarantee that one can never read an uninitialized field, therefore rule 13 ensures that all fields are assigned at least once.

All three fields in `A` are asynchronously assigned, however, only `f2` is not definitely assigned at the end of the constructor. Assigning to `f3` has an enclosing `finish`, so it is definitely assigned. Field `f1` is also definitely assigned, because it is non-final so from rule 7 it has an implicit zero field initializer. However, field `f2` is final so it does not have an implicit field initializer. Moreover, `f2` is only asynchronously assigned, and the constructor does not have to wait for this assignment to finish, thus violating rule 13. (The exact data-flow analysis to enforce rule 13 is described in Sec. 2.11.) Rule 14 is the same as in Java: a final field is assigned *at most* once (and, combined with rule 13, we know it is assigned *exactly* once).

2.10 Distributed Programming and Initialization

X10 programs can be executed on a distributed system with multiple places that have no shared memory. Objects are copied from one place to another when captured by an `at`. Copying is done by first serializing the object into a buffer, sending the buffer to the other place, and then de-serializing the buffer at the other place. As in Java, one can write custom serialization code in X10 by implementing the `CustomSerialization` interface, and defining the method `serialize():SerialData` and the constructor `this(data:SerialData)`.

Fig. 8b shows a common pitfall where a raw `this` escapes to another place, and the field assignment would have been done on a copy of `this`. We wish to de-serialize only cooked objects, and therefore rule 15 prohibits `this` to be captured by an `at`. Consequently, we also report that field `f` was not definitely assigned.

2.11 Read and Write of Fields

We now present a data-flow analysis for guaranteeing that a field is read only after it was written, and that a final field is assigned exactly once. Java performs an *intra*-procedural data-flow analysis in *constructors* to calculate when a *final* field is definitely-assigned and definitely-unassigned. In contrast, X10 performs an *inter*-procedural fixed-point data-flow analysis in all *non-escaping methods* (and constructors) to calculate when a field (*both final and non-final*) is definitely-assigned, *definitely-asynchronously-assigned*, and definitely-unassigned. The details are explained using examples (Fig. 9) by comparison with Java; the full analysis is described in X10's language specification.

X10, like Java, allows *writing* to a final field only when it is definitely-*unassigned*, and it allows *reading* from a final field only when it is definitely-*assigned*. X10 also has the same read restriction on non-final fields (recall that rule 7 adds a field initializer if the field's type has-zero).

Consider first only final fields. They are easier to type-check because they can only be assigned in constructors. X10 extends Java rules, by calculating for each non-escaping method `m` the set of final fields it reads, and calling `m` is legal only if these fields have been

```
class A {
 val a:Int;
 def this() {
  readA(); //err1
  finish {
   async {
    a = 1;
    // assigned={a}
    readA();
   }
   // asyncAssigned={a}
   readA(); //err2
  }
  // assigned={a}
  readA();
 }
 // reads={a}
 private def readA() {
  val x = a;
 }

}
```

(a)

```
class B {
 var i:Int{self!=0}, j:Int{self!=0};
 def this() {
  finish {
   asyncWriteI(); // asyncAssigned={i}
  } // assigned={i}
  writeJ();// assigned={i,j}
  readIJ();
 }
 // asyncAssigned={i}
 private def asyncWriteI() {
  async i=1;
 }
 // reads={i} assigned={j}
 private def writeJ() {
  if (i==1) writeJ(); else this.j = 1;
 }
 // reads={i,j}
 private def readIJ() {
  val x = this.i+this.j;
 }
}
```

(b)

Fig. 9. Read-Write order for fields. We infer for each method three sets: (i) fields it reads (i.e., these fields must be assigned before the method is called), (ii) fields it assigns, (iii) fields it assigns asynchronously. The data-flow maintains these three sets before and after each statement; *`assigned`* becomes *`asyncAssigned`* after an `async`, and *`asyncAssigned`* becomes *`assigned`* after a `finish`. In this example, we omitted empty sets. **Rule 16:** A field may be read only if it is definitely-assigned. **Rule 17:** A final field may be written only if it is definitely-unassigned.

definitely assigned. For example, in class `A`, method `readA` reads field `a` and therefore cannot be called before `a` is assigned (e.g., *err1*). Note that Java does not perform this check, and it is legal to call `readA` which will return the zero value of `a`. X10 also adds the notion of *definitely-asynchronously-assigned* which means a field was definitely-assigned within an `async` (so it cannot be read, e.g., *err2*), but after an enclosing `finish` it will become definitely-assigned (so it can be read). The flow maintains three sets: *reads*, *assigned*, and *asyncAssigned*. If a method reads an uninitialized field, then we add it to its *reads* set; however, if a constructor reads an uninitialized field, then it is an error. Phrased differently, the *reads* set of a constructor must be empty.

Now consider non-final fields. They can be assigned and read in methods, thus requiring a fixed-point algorithm. For example, consider method `writeJ`. Initially, *reads* is empty, while *assigned* and *asyncAssigned* are the entire set of fields. In the first iteration, we add `i` to *reads*, and when we join the two branches of the `if`, *assigned* is decreased to only `j`. The fixed-point calculation, in every iteration, increases *reads* and decreases *assigned* and *asyncAssigned*, until a fixed-point is reached.

3 Implementation

This section discusses our implementation inside the X10 compiler of the hardhat initialization rules. Our X10 code-base of more than 200K lines of code (loc) uses only 104 annotations. We give some measurements such as compilation time and annotation overhead, and conclude with two examples for `@NonEscaping` and `@NoThisAccess`.

The X10 compiler is based on the Polyglot extensible compiler framework, which includes a dataflow framework that has 1309 loc. X10 initialization rules extend this dataflow framework using two classes: one for checking definite-initialization for *local variables* (805 loc), and another for *fields* (951 loc). (The rules of local variables are simpler than those for fields because local variables do not span multiple methods and they must be assigned before use. The focus point of this paper has been object initialization, therefore these rules were not described in this paper.) The dataflow algorithm tracks for each field (or local) the flow of this information: (i) whether the field was read (to find the set of fields each non-escaping method reads), (ii) the minimal and maximal number of times it was sequential and asynchronously written (to make sure a variable is assigned before read, and that final variables are assigned exactly once). The number of times a variable is assigned is sufficient to range between 0, 1, and *more-than-one*, because the error message is the same whether a final variable was assigned twice or more. When flowing out of an `async`, sequential writes become asynchronous writes, and the opposite happens for a `finish`.

Our code-base consists of 5 major components: (i) XRX: X10 runtime and libraries, (ii) SPECjbb: SPECjbb from 2005 converted to X10, (iii) M3R: map-reduce in X10, (iv) UTS: global load balancing library, (v) MISC: those include examples from our programmer guide, our test suite, jira issues, and samples. SPECjbb and M3R are still under development and not publicly available, whereas the rest are open-source and available at `x10-lang.org` (see revision 23028 of `https://x10.svn.sf.net/svnroot/x10/trunk`).

Tab. 1 shows the compilation times broken down according to the time spent for checking fields and locals. We can see that the initialization rules take only a small fraction (0-2%) from the total compilation time, and a maximum of 3.3 seconds for the entire M3R project.

Table 1. Compilation times in milliseconds of our code-base broken down into the time spent by the initialization rules for fields and locals. We used a standard lenovo T500 laptop with 4GB of RAM and Intel Core 2 Duo processor.

	XRX	SPECjbb	M3R	UTS	MISC
Total compilation time	65,241	78,952	254,020	72,205	548,547
Time of checks for fields	156	1,649	3,330	1,272	2,862
Time of checks for locals	32	51	117	33	126

Tab. 2 shows the annotation burden in our code-base. X10 has only two possible method annotations: `@NonEscaping` and `@NoThisAccess`. Recall that all methods transitively called from a constructor are implicitly non-escaping, i.e., the user does not have to explicitly annotate them as `@NonEscaping`, however the compiler issues a warning recommending that they should be marked as such to show intent. Obviously, the number of non-escaping methods is always greater or equal to the number of `@NonEscaping` annotations. As can be seen, the annotations burden is minor: only 104 annotations in total.

Table 2. The annotation burden in our code-base

	XRX	SPECjbb	M3R	UTS	MISC
# of lines	27,153	14,603	71,682	2,765	155,345
# of files	257	63	294	14	2,283
# of constructors	276	267	401	23	1,297
# of methods	2,216	2,475	2,831	124	8,273
# of non-escaping methods	8	38	34	3	83
# of `@NonEscaping`	7	7	13	1	62
# of `@NoThisAccess`	1	0	1	0	12

Our applications only use `@NoThisAccess` twice: once in M3R to allow a subclass to determine the value of a final field of the superclass during initialization, and the second time in XRX in method `typeName()` of interface `Any` (this method may be overridden and it is often called during construction for debugging purposes).

The following example shows a common pattern for using `@NonEscaping` and a common refactoring that was done when converting Java code to X10. Class `HashMap` in Java calls `put` in two constructors: the deserialization constructor and the copy constructor (that gets a map argument and creates a copy of that map). However, `put` is not a final method and it might be useful to override it in subclasses, and therefore it cannot be called during construction. Thus, we refactored this code in X10 and called instead a non-escaping method called `putInternal` and method `put` delegates to that method:

```
public def put(k: K, v: V) { putInternal(k,v); }
@NonEscaping protected final def putInternal(k:K, v:V) { ... }
```

A similar refactoring was also done in `HashMap` for method `rehash`.

Asynchronous initialization was not used in our big applications because they predate this feature. (It is used in our smaller examples and tests more than 50 times.)

This pattern is especially useful for local variables, and more importantly, the analysis prevents bugs such as:

```
val x:Int; val y:Int;
finish { async { x = doCalculation1(); }
  y = doCalculation2(); // WRONG to use variable x here
} // OK to use variable x now
```

4 Formalism: FX10

Featherweight X10 (FX10) is a formal calculus for X10 intended to complement Featherweight Java (FJ). It models imperative aspects of X10 including the concurrency constructs `finish` and `async`. FX10 models the heart of the field initialization problem: a field can be read only after it is definitely assigned.

The basic idea behind the formalization is very straightforward. We break up the formalization into two distinct but interacting subsystems, a *type system* (Sec. 4.2) and an *effect* system (Sec. 4.3). The type system is completely standard – think the system of FJ, adapted to the richer constructs of FX10.

The effect system is built on a very simple *logic of initialization assertions*. The primitive formula $+\mathtt{x}$ ($+\mathtt{p.f}$) asserts that the variable `x` (the field `f` of `p`) is definitely initialized with a cooked object, and the formula $-\mathtt{x}$ ($-\mathtt{p.f}$) asserts that it is being initialized by a concurrent activity (and hence it will be definitely initialized once an enclosing `finish` is crossed). An *initialization formula* ϕ or ψ is simply a conjunction of such formulas $\phi \wedge \psi$. An *effects assertion* ϕ `S` ψ (for a statement `S`) is read as a partial correctness assertion: when executed in a heap H that satisfies the constraint ϕ, `S` will on termination result in a heap H' that satisfies ψ. Since we do not model `null`, our formalization can be particularly simple: variables, once initialized, stay initialized, hence H' will also satisfy ϕ (see Sec. 4.4 for a definition of heaps and when a heap satisfy ϕ).

Another feature of our approach is that, unlike Masked Types [10], the source program syntax does not permit the specification of initialization assertions. Instead we use a standard least fixed point computation to automatically decorate each method $\mathtt{def\ m}(\overline{\mathtt{x}} : \overline{\mathtt{C}})\{\mathtt{S}\}$ with pairs (ϕ, ψ) (in the free variables $\mathtt{this}, \overline{\mathtt{x}}$) such that under the assumption that all methods satisfy their corresponding assertion we can show that ϕ `S` ψ.[1] This computation must be sensitive to the semantics of method overriding, that is a method with decoration (ϕ, ψ) can only be overridden by a method with decoration (ϕ', ψ') that is "at least as strong as" (ϕ, ψ) (viz, it must be the case that ($\phi \vdash \phi'$ and $\psi' \vdash \psi$). Further, if the method is not marked `@NonEscaping`, then ϕ is required to entail $+\mathtt{this}$ (that is, `this` is cooked), and if it marked `@NoThisAccess` then ϕ, ψ cannot have `this` free.

[1] Note that this approach permits a formal `x` to a method to be completely raw (ϕ does not entail $+\mathtt{x}$ or $+\mathtt{x.f}$ for any field `f`) or partially raw (ϕ does not entail $+\mathtt{x}$ but may entail $+\mathtt{x.f}$ for some fields `f`). As a result of the method invocation the formal may become more cooked. In X10, in order for the inference to be intra-class, we require that all method parameters $\overline{\mathtt{x}}$ (except `this`) are cooked, i.e., $+\mathtt{x}_i$. In FX10 we are more relaxed and allow methods to receive and initialize raw parameters.

By not permitting the user to specify initialization assertions we make the source language much simpler than [10] and usable by most programmers. The down side is that some initialization idioms, such as cyclic initialization, are not expressible.

For reasons of space we do not include the details behind the decoration of methods with initialization assertions. We also omit many extensions (such as generics, interfaces, constraints, casting, inner classes, overloading, co-variant return types, final, field initializers etc.) that were discussed in the first half of the paper. FX10 also does not model places because the language design decision to only permit cooked objects to cross places means that the rules for `at` are routine.

We use the usual notation of $\overline{\mathtt{x}}$ to represent a vector or set of $\mathtt{x}_1, \ldots, \mathtt{x}_n$. A program `P` is a pair of class declarations $\overline{\mathtt{L}}$ (that is assumed to be global information) and a statement `S`.

4.1 Syntax

Fig. 10 shows the syntax of FX10. Expression `val x = e;S` evaluates `e`, assigns it to a new variable `x`, and then evaluates `S`. The scope of `x` is `S`.

The syntax is similar to the real X10 syntax with the following difference: FX10 does not have constructors; instead, an object is initialized by assigning to its fields or by calling non-escaping methods.

`P` ::= $\overline{\mathtt{L}}$, `S`	Program.
`L` ::= `class C extends D {` $\overline{\mathtt{F}}$; $\overline{\mathtt{M}}$ `}`	cLass declaration.
`F` ::= `var f : C`	Field declaration.
`M` ::= `G def m(`$\overline{\mathtt{x}}$ `:` $\overline{\mathtt{C}}$`) : C{S}`	Method declaration.
`G` ::= `@NonEscaping` | `@NoThisAccess`	Method modifier.
`p` ::= `l` | `x`	Path.
`e` ::= `p.f` | `new C`	Expressions.
`S` ::= `p.f = p;` | `p.m(`$\overline{\mathtt{p}}$`);` | `val x = e;S` | `finish {S}` | `async {S}` | `S S`	Statements.

Fig. 10. FX10 Syntax. The terminals are locations (`l`), parameters and `this` (`x`), field name (`f`), method name (`m`), class name (`B`,`C`,`D`,`Object`), and keywords (`new`, `finish`, `async`, `val`). The program source code cannot contain locations (`l`), because locations are only created during execution/reduction in R-NEW of Fig. 12.

4.2 Type System

The type system for FX10 checks that every parameter and variable has a type (a type is the name of a class), and that a variable of type `C` can be assigned only expressions whose type is a subclass of `C`, and can only be the receiver of invocations of methods defined in `C`. The type system is formalized along the lines of FJ. No complications are introduced by the extra features of FX10 – assignable fields, local variable declarations, `finish` and `async`. We omit details for lack of space and because they are completely routine. In the rest of this section we shall assume that the program being considered $\overline{\mathtt{L}}$, `S` is well-typed.

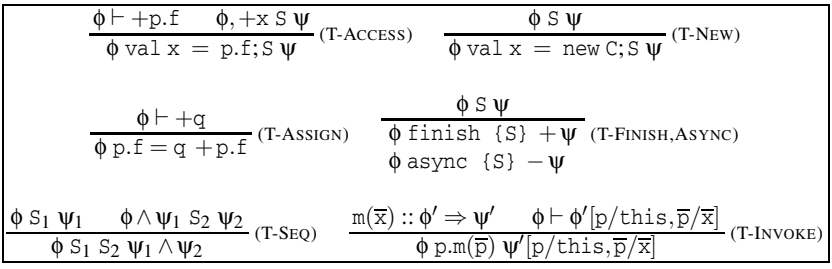


Fig. 11. FX10 Effect System (ϕ S ψ)

4.3 Effect System

We use a simple logic of initialization for our basic assertions. This is an intuitionistic logic over the primitive formulas $+\mathtt{p}$ (the variable or parameter p is initialized), $+\mathtt{p.f}$ (the field p.f is initialized), and $-\mathtt{p}, -\mathtt{p.f}$ (it is being *concurrently* initialized). We are only concerned with conjunctions and existential quantifications over these formulas: $\phi, \psi ::= \mathtt{true} \mid +\mathtt{x} \mid +\mathtt{p.f} \mid -\mathtt{p.f} \mid \phi \wedge \psi$

The notion of substitution on formulas $\phi[\overline{\mathtt{x}}/\overline{\mathtt{z}}]$ is specified in a standard fashion.

The inference relation is the usual intuitionistic implication over these formulas, and the following additional proof rules: (1) if $\phi \vdash +\mathtt{p}$ then $\phi \vdash -\mathtt{p}$; (2) if $\phi \vdash +\mathtt{p.f}$ then $\phi \vdash -\mathtt{p.f}$; (3) if $\phi \vdash +\mathtt{p}$ ($\phi \vdash -\mathtt{p}$) then $\phi \vdash +\mathtt{p.f}$ ($\phi \vdash -\mathtt{p.f}$); and (4) if the *exact* class of p is C, and C has the fields $\overline{\mathtt{f}}$, then $\phi \vdash +\mathtt{p}$ ($\phi \vdash -\mathtt{p}$) if $\phi \vdash +\mathtt{p.f}_i$ ($\phi \vdash -\mathtt{p.f}_i$), for each *i*. (We only know the *exact* class for a local p when $\mathtt{val\ p\ =\ new\ C;S}$.)

The proof rules for the judgement ϕ S ψ are given in Figure 11. They use two syntactic operations on initialization formulas defined as follows. $+\psi$ is defined inductively as follows: $+\mathtt{true} = \mathtt{true}$, $+\pm\mathtt{x} = +\mathtt{x}$, $+\pm\mathtt{p.f} = +\mathtt{p.f}$, $+(\phi \wedge \psi) = (+\phi) \wedge (+\psi)$. $-\psi$ is defined similarly: $-\mathtt{true} = \mathtt{true}$, $-\pm\mathtt{x} = -\mathtt{x}$, $-\pm\mathtt{p.f} = -\mathtt{p.f}$, $-(\phi \wedge \psi) = (-\phi) \wedge (-\psi)$.

The rule (T-ACCESS) can be read as asserting: if ϕ entails the field p.f is initialized (together with $+\mathtt{x}$ which states that x is initialized to a cooked object), we can establish that execution of S satisfies the assertion ψ then we can establish that execution of $\mathtt{val\ x\ =\ p.f;S}$ in (a heap satisfying) ϕ establishes ψ. Here we must take care to project x out of ψ since x is not accessible outside its scope S; similarly we must take care to project x out of ϕ when checking S. The rule (T-NEW) can be read in a similar way except that when executing S we can make no assumption that x is initialized, since it has been initialized with a raw object (none of its fields are initialized). Subsequent assignments to the fields of x will introduce effects recording that those fields have been initialized. The rule (T-ASSIGN) checks that q is initialized to a cooked object and then asserts that p.f is initialized to a cooked object. The rule (T-FINISH) can be understood as recording that after a finish has been "crossed" all asynchronous initializations ψ can be considered to have been performed ϕ. Conversely, the rule (T-ASYNC) states that any initializations must be considered asynchronous to the surrounding context. The rule (T-SEQ) is a slight variation of the stadard rule for sequential composition that permits ϕ to be used in the antecedent of $\mathtt{S}_2$, exploiting monotonicity of effects. Note the effects recorded for $\mathtt{S}_1\ \mathtt{S}_2$ are a conjunction of the effects recorded for $\mathtt{S}_1$ and $\mathtt{S}_2$. The rule (T-INVOKE) is routine.

As an example, consider the following classes. Assertions are provided in-line.

```
class A extends Object {
  var f:Object; var g:Object; var h:Object;
  @NonEscaping def build(a:Object) {
    // inferred decoration: phi => psi
    //   phi= +this.g, +a
    //   psi= -this.h, +this.f
    // checks phi implies +this.g
    val x = this.g;
    async { this.h = x; } // psi= -this.h
    finish {
      // checks phi implies +a
      async { this.f = a; } // psi= -this.h,-this.f
    } // psi= -this.h,+this.f
  }
}
class B extends A { e:Object; }
```

Method `build` synchronously (asynchronously) initializes fields `this.f` (`this.h`), and it assumes that `this.g` and `a` are cooked. The following statement completely initializes `b`:

```
val b = new B();
val a = new Object(); // psi= +a
b.g = a; // psi= +a,+b.g
finish {
  b.build(a); // psi= +a,+b.g,+b.f,-b.h
  async { b.e = a; } // psi= +a,+b.g,+b.f,-b.h,-b,-b.e,-b
} // psi= +a,+b
```

4.4 Reduction

A *heap* H is a mapping from a given set of locations to *objects*. An object is a pair $C(F)$ where C is a class (the exact class of the object), and F is a partial map from the fields of C to locations. We say the object `l` is *total/cooked* (written $cooked_H(\texttt{l})$) if its map is total, i.e., $H(\texttt{l}) = \texttt{C}(F) \qquad \mathrm{dom}(F) = \mathit{fields}(\texttt{C})$.

We say that a heap H *satisfies* ϕ (written $H \vdash \phi$) if the plus assertions in ϕ (ignoring the minus assertions) are true in H, i.e., if $\phi \vdash +\texttt{l}$ then `l` is cooked in H and if $\phi \vdash +\texttt{l.f}$ then $H(\texttt{l}) = \texttt{C}(F)$ and $F(\texttt{f})$ is cooked in H.

The reduction relation is described in Figure 12. An S-configuration is of the form $\texttt{S}, H$ where `S` is a statement and H is a heap (representing a computation which is to execute `S` in the heap H), or H (representing terminated computation). An E-configuration is of the form $\texttt{e}, H$ and represents the computation which is to evaluate `e` in the configuration H. The set of *values* is the set of locations; hence E-configurations of the form $\texttt{l}, H$ are terminal.

Two transition relations $\rightsquigarrow$ are defined, one over S-configurations and the other over E-configurations. For X a partial function, we use the notation $X[v \mapsto e]$ to represent the partial function which is the same as X except that it maps v to e. The rules defining these relations are standard. The only minor novelty is in how `async` is defined. The critical rule is the last rule in (R-STEP) – it specifies the "asynchronous" nature of `async`

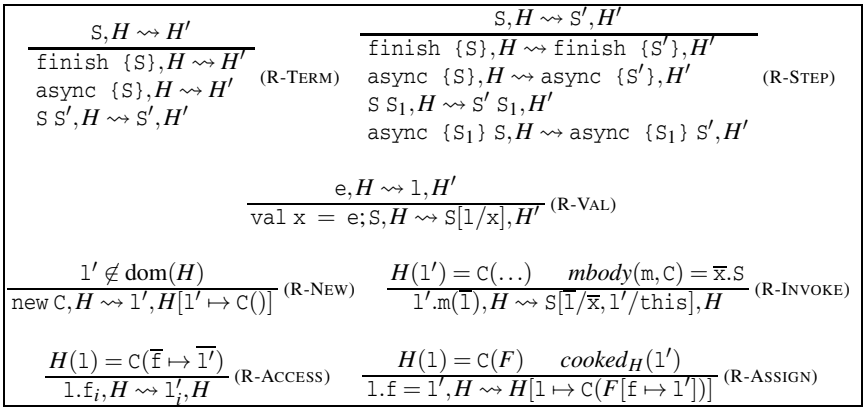


Fig. 12. FX10 Reduction Rules ($\mathtt{S},H \rightsquigarrow \mathtt{S}',H' \mid H'$ and $\mathtt{e},H \rightsquigarrow \mathtt{l},H'$)

by permitting S to make a step even if it is preceded by async {S1}. The rule (R-NEW) returns a new location that is bound to a new object that is an instance of C with none of its fields initialized. The rule (R-ACCESS) ensures that the field is initialized before it is read ($\mathtt{f}_i$ is contained in $\overline{\mathtt{f}}$).

4.5 Results

We say a heap H is *correctly cooked* (written $\vdash H$) if a field can point only to cooked objects, i.e., for every object $o = C(F)$ in the range of H and every field $f \in \mathrm{dom}(F)$ it is the case that every object $\mathtt{l} = H(F(f))$ is cooked ($cooked_H(\mathtt{l})$). We shall only consider correctly cooked heaps (valid programs will only produce correctly cooked heaps). As the program is executed, the heap monotonically becomes more and more cooked. Formally, H' is *more cooked* than H (written $H' \vdash H$) if for every $\mathtt{l} \in \mathrm{dom}(H)$, we have $H(\mathtt{l}) = \mathtt{C}(F)$, $H'(\mathtt{l}) = \mathtt{C}(F')$, and $\mathrm{dom}(F) \subseteq \mathrm{dom}(F')$.

A *heap typing* Γ is a mapping from locations to classes. H is said to be typed by Γ if for each $l \in \mathrm{dom}(H)$, the class of $H(l)$ is a subclass of $\Gamma(l)$. Since our treatment separates out effects from types, and the treatment of types is standard, we shall assume that all programs and heaps are typed.

A statement S is *closed* (written $\vdash \mathtt{S}$) if it does not contain any free variables. We say that S is *annotatable* if there exists ϕ, ψ such that ϕ S ψ can be established.[2]

We say that a program $\mathtt{P} = \overline{\mathtt{L}}\mathtt{S}$ is *proper* if it is well-typed and each method in L can be decorated with pre-post assertions (ϕ, ψ), and S is annotatable. The decorations must satisfy the property that under the assumption that every method satisfies its assertion (this is for use in recursive calls) we can establish for every method $\mathtt{def\ m}(\overline{\mathtt{x}} : \overline{\mathtt{C}})\{\mathtt{S}\}$ with assertion (ϕ, ψ) that it is the case that the free variables of ϕ, ψ are contained in $\mathtt{this}, \overline{\mathtt{x}}$, and that ϕ S ψ.

[2] An example of a statement that is *not* annotatable is val x = new C;val y = x.f;z.g = y where C has a field f. This attempts to read a field of a variable initialized with a brand-new object.

We prove the following theorems. In all these theorems the background program $\mathtt{P}$ is assumed to be proper. The first theorem is analogous to subject-reduction for typing systems.

Theorem 1. ***Preservation*** *Let* $\phi\ \mathtt{s}\ \psi, \vdash \mathtt{s}, \vdash H, H \vdash \phi$. *(a) If* $\mathtt{s}, H \rightsquigarrow H'$ *then* $\vdash H', H' \vdash H$, $H' \vdash +\psi$. *(b) If* $\mathtt{s}, H \rightsquigarrow \mathtt{s}', H'$ *then* $\vdash \mathtt{s}', \vdash H', H' \vdash H$, *there exists* ϕ', ψ' *such that* $H' \vdash \phi'$, $\phi'\ \mathtt{s}'\ \psi', \phi' \vdash \phi, \psi' \vdash \psi$.

Theorem 2. ***Progress*** *Let* $\phi\ \mathtt{s}\ \psi, \vdash \mathtt{s}, \vdash H, H \vdash \phi$. *The configuration* $\mathtt{s}, H$ *is not stuck.*

For proofs, please see associated technical report.

Because our reduction rules only allow reads from initialized fields, a corollary is that a field can only be read after it was assigned, and an attempt to read a field will always succeed.

5 Related Work

A static analysis [11], has been used to find some default value reads in Java programs, and supports our belief that default value reads can be found in real programs and should be considered errors. Our approach is stronger (detecting all errors at the expense of some correct programs) and considers additional language constructs that are not present in Java.

There has been a study on a large body [6] of Java code, showing that initialization order issues pervade projects from the real world. A bytecode verification system for Java initialization has also been explored [7].

An early work to support non-null types in Java [3] has the notion of a type constructor *raw* that can be applied to object types and means that the fields of the object (in X10 terminology) may violate the constraints in their types. Our approach permits optimization of the representation of fields whose types are very constrained, since they will never have to hold a value other than the values allowed by their type constraint.

A later work [4,10] allows to specify the time (in the type) when the object will be fully constructed. Field reference types of a partially constructed objects must be fully constructed by the same time, which allows graphs of objects to be constructed like our `proto` design. However the system is more complicated, allowing the object to become fully constructed at a given future time, instead of at the specific time when its constructor terminates.

Masked types [10] present types that describe the exact fields that have not yet been initialized. Summers and Müller [12] describe a simpler type system that is almost identical to our `proto` design, however they only treat non-null types and they allow reading a field before it was assigned. Our type system is simpler but less expressive because it cannot handle immutable cyclic structures.

There is also a time-aware type system [8] that allows the detection of data-races, and understands the concept of shared variables that become immutable only after a certain time (and can then be accessed without synchronization). The same mechanisms can also be used to express when an object becomes cooked.

Ownership types can be used to create immutable cycles [14]. This is comparable to our `proto` design because it also allows `this` to be linked from an incomplete object. However the ownership structure is used to implement a broader policy, allowing code

in the owner to use a reference to its partially constructed children, whereas we only allow code to use a reference to objects that are being partially constructed in some nesting stack frame. Our approach does not use ownership types.

6 Conclusion

The hardhat design in X10 is strict but it protects the user from error-prone initialization idioms, especially when combined with a rich type system and parallel code. This paper showed the interaction between initialization and other language features, possible pitfalls in Java, and how they can be prevented in X10. It also presented the rules of this design, the virtues of these rules, and possible design alternatives. The rules were incorporated in the open-source X10 compiler, and are being used in production code.

References

1. Bocchino Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel java. In: OOPSLA 2009, pp. 97–116. ACM, New York (2009)
2. Dean, D., Felten, E., Wallach, D.S.: Java security: From hotjava to netscape and beyond. In: IEEE Symposium on Security and Privacy, pp. 190–200 (1996)
3. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: OOPSLA 2003, pp. 302–312 (2003)
4. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA 2007, pp. 337–350 (2007)
5. Gil, J., Itai, A.: The Complexity of Type Analysis of Object Oriented Programs. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 601–634. Springer, Heidelberg (1998)
6. Gil, J.Y., Shragai, T.: Are We Ready for a Safer Construction Environment? In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 495–519. Springer, Heidelberg (2009)
7. Hubert, L., Jensen, T., Monfort, V., Pichardie, D.: Enforcing Secure Object Initialization in Java. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 101–115. Springer, Heidelberg (2010)
8. Matsakis, N.D., Gross, T.R.: A time-aware type system for data-race protection and guaranteed initialization. In: OOPSLA 2010, pp. 634–651 (2010)
9. Pugh, W.: JSR 133: Java memory model and thread specification revision (2004), http://jcp.org/en/jsr/detail?id=133
10. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL 2009, pp. 53–65 (2009)
11. Seo, S., Kim, Y., Kang, H.-G., Han, T.: A static bug detector for uninitialized field references in java programs. IEICE - Trans. Inf. Syst. E90-D, 1663–1671 (2007)
12. Summers, A.J., Müller, P.: Freedom before commitment - a lightweight type system for object initialisation. In: OOPSLA 2011 (2011)
13. Yang, X., Blackburn, S.M., Frampton, D., Sartor, J.B., McKinley, K.S.: Why nothing matters: the impact of zeroing. In: OOPSLA 2011, pp. 307–324. ACM, New York (2011)
14. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in generic java. In: OOPSLA 2010, pp. 598–617 (2010)

Structured Aliasing

Tobias Wrigstad

Uppsala University
tobias.wrigstad@it.uu.se
http://wrigstad.com

Abstract. Aliasing, mutable state and stable object identities are inherent in object-oriented programming. It is a well-known fact that this troika of features cause problems for practitioners, tool developers and formalists alike. Patterns for aliasing, and patterns for structuring object graphs exist, and manipulating aliases and managing these graphs or graph-like structures are among the most frequent operations in object-oriented programming. Yet, mainstream languages provide only low-level support for these operations in reading and writing of variables. The bookkeeping, making sure graph structures are preserved, temporary aliases deleted, etc. is left to the programmer.

In this talk, I will review some of my work on managing aliases in object-oriented systems, and talk about some recent efforts to unify these approaches to provide what we could call a theory and practise of structured aliasing for object-oriented programming.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, p. 232, 2012.

Pause 'n' Play: Formalizing Asynchronous C♯

Gavin Bierman[1], Claudio Russo[1], Geoffrey Mainland[1],
Erik Meijer[2], and Mads Torgersen[3]

[1] Microsoft Research
[2] Microsoft Corp. and TU Delft
[3] Microsoft Corporation
{gmb,crusso,gmainlan,emeijer,madst}@microsoft.com

Abstract. Writing applications that connect to external services and yet remain responsive and resource conscious is a difficult task. With the rise of web programming this has become a common problem. The solution lies in using asynchronous operations that separate issuing a request from waiting for its completion. However, doing so in common object-oriented languages is difficult and error prone. Asynchronous operations rely on callbacks, forcing the programmer to cede control. This inversion of control-flow impedes the use of structured control constructs, the staple of sequential code. In this paper, we describe the language support for asynchronous programming in the upcoming version of C♯. The feature enables asynchronous programming using structured control constructs. Our main contribution is a precise mathematical description that is abstract (avoiding descriptions of compiler-generated state machines) and yet sufficiently concrete to allow important implementation properties to be identified and proved correct.

1 Introduction

Mainstream programmers are increasingly adopting asynchronous programming techniques once the preserve of hard-core systems programmers. This adoption is driven by a variety of reasons: hiding the latency of the network in distributed applications; maintaining the responsiveness of single-threaded applications or simply avoiding the resource cost of creating too many threads. To facilitate this programming style, operating systems and platforms have long provided non-blocking, asynchronous alternatives to possibly blocking, synchronous operations. While these have made asynchronous programming possible they have not made it easy.

The basic principle behind these asynchronous APIs is to decompose a synchronous operation that combines issuing the operation with a blocking wait for its completion, into a non-blocking initiation of the operation, that immediately returns control, and some mechanism for describing what to do with the operation's result once it has completed. The latter is typically described by a callback—a method or function. The callback is often supplied with the initiation as an additional argument. Alternatively, the initiation can return a handle which the client can use to selectively register an asynchronous callback or (synchronously) wait for the operation's result.

Whatever the mechanism, the difficulty with using these APIs is fundamentally this: to transform a particular synchronous call-site into an asynchronous call-site requires the programmer to represent the continuation of the original site as a callback. Moreover, for this callback to resume from where the synchronous call previously returned, it must preserve all of the state pertinent to the continuation of the call. Some aspects of the state

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 233–257, 2012.

will be explicitly available (such as the values of local variables), but other aspects may not be. A prime example is the remainder of the current call stack. For languages that do not provide support for first-class continuations, like Java and C♯, accounting for this state often requires a manual transformation to continuation-passing-style of not only the enclosing method, but also all of its callers. Once reified as an explicit continuation, the state of a computation can be saved at the initiation of an asynchronous operation and restored on its completion by supplying it with a result.

The upcoming version of C♯ (and Visual Basic) features dedicated linguistic support for asynchronous programming that removes the need for explicit callbacks. C♯ 5.0 allows certain methods to pause and then later resume their computation, without blocking, at explicitly marked code points. The basic idea is to allow a method, designated as asynchronous, to *await* the completion of some other event, not by blocking its executing thread, but by *pausing* its own execution and releasing its thread to do further work. The caller of the paused method then receives a *task* representing the method's future result and is free to proceed. Subsequent completion of the awaited event causes the paused method to resume *playing* from where it left off. Since its original thread has carried on, the resumed method is played on some available thread. Depending on run-time context, this thread may be drawn from the .NET thread pool, or it may be the same, issuing thread but at a later opportunity (e.g., the resumed method might be re-enqueued in the user interface's message loop). The events that can be awaited are typically tasks returned by nested calls to asynchronous methods. They can also, more generally, belong to any (user-defined) *awaitable* type or primitive implementations provided by the framework. The aim of these new features is to make it easy to write asynchronous methods, without having to resort to continuation-passing-style and its debilitating inversion of control flow.

As we shall see, the new asychronous features in C♯ 5.0 are quite subtle. Current, draft Microsoft specifications [16] describe the features using precise prose and by example, giving illustrative source-to-source translations from C♯ 5.0 to ordinary C♯ 4.0. Unfortunately, the translation is intricate—it compiles source to optimized, finite state machines—so its output is both verbose and difficult to comprehend. We believe a formal, mathematical approach can yield both a precise foundation for researchers, but also a better mental model for developers and compiler writers to justify the correctness of their translation. The primary contribution of this paper is to provide such a model: a direct, operational semantics of the feature in a representative fragment of C♯ 5.0. Our semantics both capture the intent of the feature and explain its performance-driven limitations, without appealing to low-level compiler output.

The paper is structured as follows. §2 gives an informal yet precise description of the feature. §2.1 presents a realistic example, re-coding a non-trival synchronous method to an asynchronous one, first using the feature, then adding concurrency and finally contrasting with an equivalent, hand-crafted implementation. §3 formally describes our core fragment of C♯ 5.0 and presents both a type system and an operational semantics. §4 sketches some of the correctness properties that our formalization satisfies. §5 presents some extensions to our basic setting; in §5.1 we show how to develop our operational semantics to be less abstract, and much closer to the implementation but still without having to resort to a finite state machine translation. In §5.2 we show how to extend our formalization to capture the awaitable pattern. §6 surveys briefly related work on asynchronous programming; and §7 presents conclusions and some future work.

2 Background: Async C♯ Extensions in a Nutshell

Syntactically, the additions to C♯ are surprisingly few: one new modifier **async** to mark a member as asynchronous and one new *expression*, **await** e, for awaiting the result—control, a value or exception—of some *awaitable* expression. An **await** expression can also be used as a statement, **await** e;, discarding its value. The **async** modifier can be placed on methods (excluding iterators) and some other method-like constructs (anonymous, first-class methods, i.e., lambdas and delegates). An **await** expression can only appear in an **async** method; other occurrences are static errors.

Statically, an **async** method must have a *taskable* return type of Task<σ>, Task or **void**. The **return** *statements* of an **async** method with return type Task<σ> may only return values of type σ, never Task<σ>! The return statements in other **async** methods may only return control (but never a value).

The argument, e, of an **await** e expression must have an *awaitable* type. The concept of awaitable type is defined by a *pattern* (of available methods). A type is awaitable when it statically supports a GetAwaiter() instance method that returns some *awaiter* type (possibly the same type). In turn, the awaiter type must support:

- a boolean instance property IsCompleted testing if the awaiter has a result now.
- a **void**-returning instance method OnCompleted(a), accepting a callback of delegate type Action.[1] Action a is a *one-shot* continuation; calling a() resumes the awaiting method. The action should be invoked *at most once*, on completion.
- a τ-returning instance method GetResult() for retrieving the result of a completed awaiter. GetResult() should either return control, some stored value, or throw some stored exception.

If the return type of GetResult is τ, then expression **await** e is an *expression* of type τ. All these operations should be (essentially) *non-blocking*.

Crucially, the types Task<σ> and Task are awaitable, allowing **async** methods to await the tasks of *nested* **async** method calls. A caller can use a returned task just like any other task (asynchronously awaiting its result, synchronously waiting for its completion or by registering an asynchronous callback). Asynchronous methods that return **void** cannot be awaited; such methods are intended for 'fire-and-forget' scenarios.

Dynamically, an **async** method executes like an ordinary method until it encounters an **await** on some value. If the value's awaiter is complete, the method continues executing using the result of the awaiter as the result of the await expression. If the awaiter is incomplete, the method registers its continuation as a callback on the awaiter and then suspends its execution. Execution will resume with the result of the awaiter, on some thread,[2] when the callback is invoked. Invoking an async method allocates a fresh, incomplete task, representing this invocation, immediately enters the method (on the caller's thread) and executes it until it encounters its first await on an incomplete awaiter. Exiting from an async method, either via return or throwing an exception, stores the result in its task, thus *completing* it. The first suspension of an async method call returns its incomplete task (or void) to its caller. If the call exits without ever suspending, it simply returns its completed task.

[1] Defined as **delegate void** Action().

[2] We are deliberately vague here; the awaiter is free to choose how the method is resumed, catering for different behaviours in, for example, single- and multi-threaded applications.

The operational semantics is deliberately designed to minimize context switches. Continuing when an awaiter is already complete ensures methods only suspend when necessary. Dually, allowing an **async** method call to begin execution on its caller's thread gives the method the opportunity to enter-and-exit quickly when possible, without imposing the cost of a context switch just to get running in the first place. This design choice morally obliges the method not to block nor even spend too much time before ceding its caller's thread (by suspension).

2.1 Example

To both illustrate and motivate the feature we present an example adapted from the Async CTP.[3] Consider the following synchronous copy function, which incrementally copies an input stream to an output stream in manageable chunks.

```
public static long CopyTo(Stream src, Stream dst) {
  var buffer = new byte[0x1000]; int bytesRead; long totalRead = 0;
  while ((bytesRead = src.Read(buffer, 0, buffer.Length)) > 0) {
    dst.Write(buffer, 0, bytesRead);
    totalRead += bytesRead; }
  return totalRead; }
```

Depending on their receivers, the calls to `Stream` methods `Read` and `Write` may well be blocking IO operations. Since this method could spend much of its time blocked, one might prefer an asynchronous variant. One way to achieve this is by replacing the synchronous calls to `Read` and `Write` with their asynchronous counterparts:

```
Task<int> ReadAsync(byte[] buffer, int offset, int count);
Task WriteAsync(byte[] buffer, int offset, int count);
```

The asynchronous variants initiate an asynchronous operation and immediately return a task representing its completion. Method `ReadAsync` begins a read and immediately returns a `Task<int>` that, once completed, will record the number of bytes actually read. The method `WriteAsync` begins a write and returns a `Task` that just tracks its completion. Tasks are completed at most once with some result. The result may be a value or some exception.

Asynchronous clients can register zero or more callbacks on a task, to be executed (on some thread) once the task has completed with a result, e.g.:

```
Task<int> rdTask = src.ReadAsync(buffer, 0, buffer.Length);
rdTask.ContinueWith((Task<int> completedRdTask) => {
  int bytesRead = completedRdTask.Result; /* won't block */
  dst.WriteAsync(buffer, 0, bytesRead); });
// do some work now
```

Since `ContinueWith` is also non-blocking, the client will quickly proceed with its work. The call to property `completedRdTask.Result` in the callback is guaranteed not to block because its task (aliasing `rdTask`) must, by causality, already be completed.

Synchronous clients can also access a task's `Result` property or just `Wait()` for its completion; these calls will block until or unless the task has completed:

[3] http://msdn.microsoft.com/vstudio/async

```
Task<int> rdTask = src.ReadAsync(buffer, 0, buffer.Length);
// do some work now
int bytesRead = rdTask.Result; /* may block */
Task wrTask = dst.WriteAsync(buffer, 0, bytesRead);
// do some more work now
wrTask.Wait(); /* may block */
```

Now let us show how C♯ 5.0 enables the simple implementation of an asynchronous version of `CopyTo`. First we mark the method as **`async`**, and then simply await the results of `src.ReadAsync` and `dst.WriteAsync`. Otherwise *the code remains the same*.

```
public static async Task<long> CopyToAsync(Stream src, Stream dst) {
  var buffer = new byte[0x1000]; int bytesRead; long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buffer,0,buffer.Length)) > 0) {
    await dst.WriteAsync(buffer, 0, bytesRead);
    totalRead += bytesRead; }
  return totalRead; }
```

Awaiting the task returned by `ReadAsync` pauses the method unless the read has completed; when play is resumed, the await expression extracts the integer value of the task. `WriteAsync` returns a non-generic `Task`; the await statement pauses the method unless the write has completed; when it is played again, execution proceeds from the next statement. Note that the return type of our asynchronous method is `Task<long>` even though its body **`return`**s a **`long`**. Clearly, this is no ordinary **`return`** statement.

Though the code is almost identical to the original, the behaviour and resource consumption is quite different. A call to `CopyTo` will repeatedly block (in the kernel) on each call to `Read` and `Write`, tying up the resources dedicated to that thread. A call of `CopyToAsync`, on the other hand, will never block; instead, each continuation of an await will be executed on demand, on some available thread in the .NET thread pool.[4]

`CopyToAsync`'s reads and writes are still being executed sequentially, but from different threads rather than a single one, so we would not expect to gain any performance from the asynchronous implementation. Indeed, given the additional scheduling and compilation overheads, the synchronous `CopyTo` is likely to execute faster.

However, we are now in a good position to *overlap* the last write with the next read, leading to this potentially faster, concurrent implementation:

```
public static async Task<long> CopyToConcurrent(Stream src, Stream dst) {
  var buffer = new byte[0x1000]; var oldbuffer = new byte[0x1000];
  int bytesRead; long totalRead = 0; Task lastwrite = null;
  while ((bytesRead = await src.ReadAsync(buffer,0,buffer.Length)) > 0) {
    if (lastwrite != null) await lastwrite; // wait later
    lastwrite = dst.WriteAsync(buffer, 0, bytesRead); // issue now
    totalRead += bytesRead;
    { var tmp = buffer; buffer = oldbuffer; oldbuffer = tmp; }; }
  if (lastwrite != null) await lastwrite;
  return totalRead; }
```

In order to achieve this, we exploit the ability to separate the initiation of a task from the act of awaiting its completion, so that we can issue the next read *during* the last

[4] If invoked from a user interface thread, each continuation will be scheduled on that thread's event queue.

write. The example illustrates the crucial advantage of allowing asynchronous methods to return incomplete, concurrently executing tasks, not just completed results. Though we emphasize concurrency, the reads and writes could also be executing in parallel, depending on the underlying streams.

In comparison, TaskJava [8] also provides constructs to avoid inversion of control while programming with asynchronous APIs. Like C♯ 5.0, TaskJava compiles straight-line code to a state machine, but its syntax is slightly more heavyweight and requires explicit calls to an event scheduler. TaskJava does not make a distinction between invoking and awaiting an asynchronous operation, so although it presents a pleasant programming model for those who wish to invoke asynchronous APIs sequentially, it is of no use to a programmer who must write code that executes multiple operations concurrently, like `CopyToConcurrent`. C♯ 5.0 does not sacrifice concurrency for convenience.

To appreciate the concision of `CopyToAsync`, let us contrast it with a representative hand-crafted version of `CopyToAsync`, `CopyToManual`, written in C♯ 4.0. Actually, this code is very close to the decompiled code emitted by the C♯ 5.0 compiler and described in the feature documentation. As mentioned earlier, the aim of our work is to eliminate the need to understand this compilation strategy.

```
public static Task<long> CopyToManual(Stream src, Stream dst) {
 var tcs = new TaskCompletionSource<long>(); // tcs.Task new & incomplete
 var state = 0; TaskAwaiter<int> readAwaiter; TaskAwaiter writeAwaiter;
 byte[] buffer = null; int bytesRead = 0; long totalRead = 0;
 Action act = null; act = () => {
  while (true) switch (state++) {
   case 0: buffer = new byte[0x1000]; totalRead = 0; continue;
   case 1: readAwaiter=src.ReadAsync(buffer,0,buffer.Length).GetAwaiter();
           if (readAwaiter.IsCompleted) continue; // goto post-read
           else { readAwaiter.OnCompleted(act); return;} // suspend at 2
   case 2: if ((bytesRead = readAwaiter.GetResult()) > 0) {
             writeAwaiter=dst.WriteAsync(buffer,0,bytesRead).GetAwaiter();
             if (writeAwaiter.IsCompleted) continue; // goto post-write
             else { writeAwaiter.OnCompleted(act); return;} // suspend at 3
            } else { state = 4; continue;} // goto post-while
   case 3: writeAwaiter.GetResult();
           totalRead += bytesRead;
           state = 1; continue; // goto pre-while
   case 4: tcs.SetResult(totalRead); // complete tcs.Task & "return"
           return; // exit machine
 }}; // end of act delegate
 act(); // start the machine on this thread
 return tcs.Task; } // on first suspend or exit from machine
```

Without going into too many details, notice how the control flow has been obscured by encoding the continuation of each await as states (here 2 & 3) of a finite state machine. The original locals, arguments and internal state of the method are (implicitly) allocated on the heap. (Note that C♯ lambdas such as `act` close over L-values, not R-values, automatically placing them on the heap; updates to those locations persist across lambda invocations.) State 0 is the initial state that sets up locals; state 1 is the **`while`** loop header, states 2 and 3 are the continuations of the **`await`** statements; state 4 is the final state and the continuation of the original **`while`** statement. State 4 exits the machine,

setting the result in the task held by shared variable `tcs` (completing the task). The finite state machine only suspends (by calling **`return`** without completing the task) in states 2 and 3 (just after an await); the other states encode internal control flow points.

3 Formalization: Featherweight C♯ 5.0

In the rest of the paper we study the essence of the new asynchronous features of C♯. To do so we take a formal, mathematical approach and define an idealized fragment, Featherweight C♯ 5.0, or $\mathsf{FC}_5^\sharp$ for short. Whilst $\mathsf{FC}_5^\sharp$ programs remain syntactically valid C♯, it is a heavily restricted fragment—any language feature that is not needed to demonstrate the essence of the asynchronous features has been removed, and the resulting fragment has been further refactored to allow for a more succinct presentation.

As the new asynchronous features predominantly affect the control flow of C♯ programs, most of our attention is on the operational semantics. In contrast, other minimal fragments such as, for example, Featherweight Java [13], and Classic Java [10], are primarily concerned with typing issues. The asynchronous features in C♯ 5.0 have almost no impact on the type system. Consequently we have stripped the type system of $\mathsf{FC}_5^\sharp$ to the core: we have only simple non-generic classes, some value types and no subtyping at all! However, we emphasize that our formalization exposes enough of the inner workings of C♯ 5.0 to allow the reader to reason about how aspects of the language, like the split between invocation and awaiting, affect the concurrent execution of multiple threads of control. There is a danger in cutting out too much of a language during formalization. For example, Fischer et al. [8] give a semantics for CJT, a simplified version of TaskJava, that avoids a heap at the expense of any ability to model communication between tasks, including a spawned thread signaling completion to its parent. In contrast, our semantics models what we claim to be the essential features of task-based programming: concurrent execution of multiple, *communicating* tasks that are invoked by the same thread of control.

$\mathsf{FC}_5^\sharp$ programs and types:

$p ::= \overline{cd}\ mb$	Program
$cd ::= \texttt{public class C \{}\overline{fd}\ \overline{md}\texttt{\}}$	Class declaration
$fd ::= \texttt{public}\ \sigma\ f\texttt{;}$	Field declaration
$md ::= \texttt{public}\ \phi\ m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)}\ mb \mid \texttt{async public}\ \psi\ m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)}\ mb$	Method declaration
$mb ::= \texttt{\{}\ \overline{\sigma}\ \overline{x}\texttt{;}\overline{s}\ \texttt{\}}$	Method body
$\phi ::= \sigma \mid \texttt{void}$	Return type
$\sigma, \tau ::= \gamma \mid \rho$	Type
$\gamma ::= \texttt{bool} \mid \texttt{int}$	Value type
$\rho ::= \texttt{C} \mid \texttt{Task<}\sigma\texttt{>}$	Reference type
$\psi ::= \texttt{Task<}\sigma\texttt{>}$	Taskable return type

Our formalization makes heavy use of the Featherweight Java [13] overbar notation, i.e., we write $\overline{x}$ for a possibly empty sequence $x_1, \ldots x_n$. We write the empty sequence as ϵ. We abbreviate operations on pairs of sequences, writing for example $\overline{\sigma}\ \overline{x}$ for the sequence $\sigma_1\ x_1, \ldots, \sigma_n\ x_n$, similarly $\overline{\sigma}\ \overline{x}\texttt{;}$ for the sequence of variable declarations $\sigma_1\ x_1\texttt{;} \ldots \sigma_n\ x_n\texttt{;}$ and finally $f(\overline{\sigma})$ for the sequence $f(\sigma_1), \ldots, f(\sigma_n)$.

A $\mathsf{FC}_5^\sharp$ program consists of a collection of class declarations and a single method body (C♯'s `main` method). A $\mathsf{FC}_5^\sharp$ class declaration **`public class`** `C` $\{\overline{fd}\ \overline{md}\}$ introduces a class `C`. We repeat that $\mathsf{FC}_5^\sharp$ does not support any form of subtyping so class declarations do not specify a superclass. This is a valid declaration in full C♯ as all classes inherit from **`object`** by default, but we do not even support the **`object`** class in $\mathsf{FC}_5^\sharp$. Subtyping and inheritance are orthogonal to the new features in C♯ 5.0 and so we removed them from our fragment to concentrate solely on the support for asynchronous programming.[5] The class `C` has fields $\overline{f}$ with types $\overline{\sigma}$ and a collection of methods $\overline{md}$.

Method declarations can be either synchronous or asynchronous. A synchronous method **`public`** $\phi\ m(\overline{\sigma}\ \overline{x})\,mb$ declares a public method m with return type ϕ, formal parameters $\overline{x}$ of type $\overline{\sigma}$ and a method body mb. Methods may be **`void`**-returning, i.e., they return control not a value. Method bodies are constrained to be of a particular form: $\overline{\sigma}\ \overline{x}\texttt{;}\overline{s}$, i.e., they must declare all their local variables $\overline{x}$ at the start of the method, and then contain a sequence of statements $\overline{s}$.

An asynchronous method is marked with the **`async`** keyword and is syntactically the same as a synchronous method, although it is type checked differently. The return type of an asynchronous method must be of a so-called taskable type. For $\mathsf{FC}_5^\sharp$ this means it must be of the form `Task<`σ`>`. C♯ 5.0 also classifies the non-generic class `Task` and **`void`** as taskable return types as discussed in §2.

$\mathsf{FC}_5^\sharp$ types are a simple subset of the C♯ types. Note that $\mathsf{FC}_5^\sharp$ does not support user-defined generics; again these are orthogonal to asynchrony and have been removed. For simplicity, we assume that `Task<`σ`>` is the only generic type.

$\mathsf{FC}_5^\sharp$ expressions and statements:

$e ::=$	Expressions
c	Constant (boolean b, integer i or **`null`**)
$x \oplus y$	Built-in operator
x	Variable
$x\texttt{.}f$	Field access
$x\texttt{.}m\texttt{(}\overline{y}\texttt{)}$	Method invocation
`new` `C()`	Object creation
`await` x	Await expression
`Task.AsyncIO<`γ`>()`	Async primitive
$s, t ::=$	Statement
x`=`e`;`	Assignment statement
`if` `(`x`)` $\{\overline{s}\}$ **`else`** $\{\overline{t}\}$	Conditional statement
`while` `(`x`)` $\{\overline{s}\}$	Iteration
$x\texttt{.}f$ `=` y`;`	Field assignment statement
$x\texttt{.}m\texttt{(}\overline{y}\texttt{);}$	Method invocation statement
`return``;`	Return statement
`return` x`;`	Return value statement

$\mathsf{FC}_5^\sharp$ expressions are restricted to a form that we call *statement normal form (SNF)*. SNF forces all subexpressions to be named; i.e., all subexpressions are simply variables. SNF

[5] The extensions to support single inheritance, overloading, constructor methods and many of the complications of the full C♯ type system have appeared elsewhere [2,3].

is the natural analogue to the A-normal-form popular in functional languages [9]. This regularity makes the presentation of the operational semantics (and the type system) much simpler at no cost to expressivity.

FC$_5^\sharp$ expressions include constants, c, which can be an integer, $\underline{i}$, a boolean, $\underline{b}$, or the literal `null`. We assume a number of built-in primitive operators, such as `==`, `<`, `>` and so on. In the grammar we write $x \oplus y$, where $\oplus$ denotes an instance of one of these operators. We do not specify operators further as their meaning is clear. We assume that x, y, z range over variable names, f ranges over field names and m ranges over method names. We assume that the set of variables includes the special variable `this`, which cannot be used as a formal parameter of a method declaration or declared as a local.

FC$_5^\sharp$ supports awaitable expressions, written **`await`** x. To get things off the ground we assume an in-built asynchronous method `Task.AsyncIO<`γ`>()` that spawns a thread and immediately returns a task. The thread may complete the task, depending on the scheduler, with some result of value type γ.

FC$_5^\sharp$ statements are standard. In what follows we assume that FC$_5^\sharp$ programs are well-formed, e.g., the identifier `this` does not appear as a formal parameter, all control paths in a method body contain a **`return`** statement, etc. These conditions can be easily formalized and are identical to restrictions on earlier fragments of C♯ but we suppress the details for lack of space. The only new well-formedness condition is that **`await`** expressions are only allowed to appear inside *asynchronous* method declarations.

We assume that a correct program induces a number of utility functions that we will use in the typing rules. First, we assume the partial function *ftype*, which is a map from a type and a field name to a type. Thus $ftype(\sigma, f)$ returns the type of field f in type σ. Second, we assume a partial function *mtype* that is a map from a type and a method name to a *type signature*. For example, we write $mtype(\mathtt{C}, m) = (\overline{\tau}) \to \phi$ when class `C` contains a method m with formal parameters of type $\overline{\tau}$ and return type ϕ.

The type system for full C♯ is actually a bidirectional type system [18] consisting of two typing relations: a type conversion relation and a type synthesis relation [3], along with a number of conversion (subtyping) judgements. However, the extreme parsimony of FC$_5^\sharp$ means that we have no subtyping judgements, and we need only a single judgement for type checking an expression. The judgement is written $\Gamma \vdash e : \sigma$ where Γ is a function from variables to types. We extend the overbar notation and write $\Gamma \vdash \overline{e} : \overline{\sigma}$ to mean the judgements $\Gamma \vdash e_1 : \sigma_1, \ldots, \Gamma \vdash e_n : \sigma_n$.

FC$_5^\sharp$ expression type checking:

$$\text{[C-Bool]}\frac{}{\Gamma \vdash \underline{b} : \mathtt{bool}} \quad \text{[C-Int]}\frac{}{\Gamma \vdash \underline{i} : \mathtt{int}} \quad \text{[C-Null]}\frac{}{\Gamma \vdash \mathtt{null} : \rho}$$

$$\text{[C-Op]}\frac{\Gamma \vdash x : \sigma_0 \qquad \Gamma \vdash y : \sigma_1 \qquad \oplus : \sigma_0 \times \sigma_1 \to \tau}{\Gamma \vdash x \oplus y : \tau}$$

$$\text{[C-New]}\frac{}{\Gamma \vdash \mathtt{new\ C()} : \mathtt{C}} \quad \text{[C-Var]}\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \text{[C-Field]}\frac{ftype(\sigma, f) = \tau}{\Gamma, x : \sigma \vdash x\,.\,f : \tau}$$

$$\text{[C-MethInv]}\frac{mtype(\sigma_0, m) = (\overline{\tau}) \to \sigma_1 \qquad \Gamma, x : \sigma_0 \vdash \overline{y} : \overline{\tau}}{\Gamma, x : \sigma_0 \vdash x\,.\,m(\overline{y}) : \sigma_1}$$

$$\text{[C-Await]}\frac{}{\Gamma, x : \mathtt{Task<}\sigma\mathtt{>} \vdash \mathtt{await}\ x : \sigma} \quad \text{[C-IO]}\frac{}{\Gamma \vdash \mathtt{Task.AsyncIO<}\gamma\mathtt{>()} : \mathtt{Task<}\gamma\mathtt{>}}$$

Most of the type checking rules are quite standard, but there are two new rules for dealing with asynchronous methods. Rule [C-Await] states that if x is of type `Task<`σ`>` then awaiting x results in a value of type σ. As discussed earlier, Rule [C-IO] states that `Task.AsyncIO<`γ`>()` returns a value of type `Task<`γ`>`.

FC$^\sharp_5$ statement type checking:

$$\text{[C-Asn]}\ \frac{\Gamma, x{:}\sigma \vdash e{:}\sigma}{\Gamma, x{:}\sigma \vdash x = e;{:}\phi} \qquad \text{[C-Cond]}\ \frac{\Gamma, x{:}\texttt{bool} \vdash \overline{s}{:}\phi \qquad \Gamma, x{:}\texttt{bool} \vdash \overline{t}{:}\phi}{\Gamma, x{:}\texttt{bool} \vdash \texttt{if (}x\texttt{) \{}\overline{s}\texttt{\} else \{}\overline{t}\texttt{\}}{:}\phi}$$

$$\text{[C-While]}\ \frac{\Gamma, x{:}\texttt{bool} \vdash \bar{s}{:}\phi}{\Gamma, x{:}\texttt{bool} \vdash \texttt{while (}x\texttt{) \{}\overline{s}\texttt{\}}{:}\phi}$$

$$\text{[C-FAsn]}\ \frac{\mathit{ftype}(\sigma_0, f) = \sigma_1 \qquad \Gamma, x{:}\sigma_0 \vdash y{:}\sigma_1}{\Gamma, x{:}\sigma_0 \vdash x\texttt{.}f\texttt{=}y\texttt{;}{:}\phi}$$

$$\text{[C-MInv]}\ \frac{\mathit{mtype}(\sigma_0, m) = (\overline{\tau}) \to \texttt{void} \qquad \Gamma, x{:}\sigma_0 \vdash \overline{y}{:}\overline{\tau}}{\Gamma, x{:}\sigma_0 \vdash x\texttt{.}m\texttt{(}\overline{y}\texttt{)}\,\texttt{;}{:}\phi}$$

$$\text{[C-Return]}\ \frac{}{\Gamma \vdash \texttt{return;}{:}\texttt{void}} \qquad \text{[C-ReturnExp]}\ \frac{}{\Gamma, x{:}\sigma \vdash \texttt{return}\ x\,\texttt{;}{:}\sigma}$$

As for full C$^\sharp$, we give type checking rules for statements; the judgement is written $\Gamma \vdash s{:}\phi$. The key rules are [C-ReturnExp] that asserts that the statement **`return`** x`;` is of return type σ if x is of type σ and [C-Return] that asserts that the statement **`return`**`;` is of return type **`void`**. In other words, the role of the type ϕ in the judgement $\Gamma \vdash s{:}\phi$ is to check any return statement. Again, we adopt an overbar notation and write $\Gamma \vdash \overline{s}{:}\phi$ to denote the judgements $\Gamma \vdash s_1{:}\phi, \ldots, \Gamma \vdash s_n{:}\phi$.

FC$^\sharp_5$ method and class typing (rule for programs omitted due to space):

$$\text{[Class-OK]}\ \frac{\texttt{C} \vdash \overline{\mathit{md}}\ \mathbf{ok}}{\vdash \texttt{public class C \{}\overline{\mathit{fd}}\ \overline{\mathit{md}}\ \texttt{\}}\ \mathbf{ok}}$$

$$\text{[Meth-OK]}\ \frac{\overline{x}{:}\overline{\sigma}, \overline{y}{:}\overline{\tau}, \texttt{this:C} \vdash \overline{s}{:}\phi \qquad \forall e \in \overline{s}, e \neq \texttt{await \_}}{\texttt{C} \vdash \texttt{public}\ \phi\ m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)\{}\overline{\tau}\ \overline{y}\texttt{;}\overline{s}\texttt{\}}\ \mathbf{ok}}$$

$$\text{[AsyncMeth-OK]}\ \frac{\overline{x}{:}\overline{\sigma}, \overline{y}{:}\overline{\tau}, \texttt{this:C} \vdash \overline{s}{:}\sigma_0}{\texttt{C} \vdash \texttt{async public Task<}\sigma_0\texttt{>}\ m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)\{}\overline{\tau}\ \overline{y}\texttt{;}\overline{s}\texttt{\}}\ \mathbf{ok}}$$

Rule [Class-OK] asserts that a class declaration is well-typed provided that all its method declarations are well-typed. Rule [Meth-OK] asserts that the (synchronous) method declaration **`public`** $\phi\ m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)\{}\overline{\tau}\ \overline{y}\texttt{;}\overline{s}\texttt{\}}$ is well-typed in class `C` provided that the statements $\overline{s}$ can be typed at return type ϕ in the context $\overline{x}{:}\overline{\sigma}, \overline{y}{:}\overline{\tau}$, **`this`**`:C`. Moreover, $\overline{s}$ cannot contain **`await`**. Rule [AsyncMeth-OK] asserts that the asynchronous method **`async public`** `Task<`σ_0`>` $m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)\{}\overline{\tau}\ \overline{y}\texttt{;}\overline{s}\texttt{\}}$ is well-typed if the statements $\overline{s}$ can be typed at return type σ_0 (not `Task<`σ_0`>`) in the context $\overline{x}{:}\overline{\sigma}, \overline{y}{:}\overline{\tau}$, **`this`**`:C`.

We assume two methods on the `Task<`σ`>` type, `Result` and `GetResult`, which both take no argument and return a value of type σ, i.e., $\mathit{mtype}(\texttt{Task<}\sigma\texttt{>}, \_) = () \to \sigma$. Both these methods return the result of a complete task object, but will differ operationally on an incomplete task. In C$^\sharp$, `Result` is actually a method-like *property*.

3.1 Operational Semantics

The key contribution of this paper is a precise description of the operational behaviour of the new asynchronous features in C♯. The syntactic restrictions of $\mathsf{FC}_5^\sharp$ mean that the operational semantics can be given as single-step transition rules between configurations.

A *heap*, H, is a partial map from an object identifier (ranged over by o) to a heap object. A *heap object* can be one of three forms: $\langle \texttt{C}, FM \rangle$ denotes a non-task object of class `C` with a *field map*, FM, which is a partial map from fields f to values. A task heap object is either of the form $\langle \texttt{Task<}\sigma\texttt{>}, \mathsf{running}(\overline{F}) \rangle$ or $\langle \texttt{Task<}\sigma\texttt{>}, \mathsf{done}(v) \rangle$. We explain these forms later. A *value*, v is either a constant, c, or an object identifier (the *address* of an object in the heap).

A *frame*, F, is written $\langle L, \overline{s} \rangle^\ell$ and consists of a locals stack, L, and a sequence of statements, $\overline{s}$, along with a frame label, ℓ. A *locals stack* is a partial map from local variables to values. A *frame label*, ℓ, is either `s` to denote a synchronous frame, or $\mathtt{a}(o)$ for an asynchronous frame whose associated `Task` is stored at heap address o. A *frame stack*, FS, is essentially a list of frames. An empty frame stack is written ϵ, and we write $F \circ FS$ to denote a frame stack whose head is a frame F and tail is the frame stack FS. A process, P, is a collection of frame stacks, written $\{FS_1, \ldots, FS_n\}$.

We factor the transition rules into three relations describing the small step evaluation of frames (method bodies), frame stacks (corresponding to individual threads) and collections of frame stacks (corresponding to a process, i.e., a pool of threads mutating a shared heap). Thus, a frame configuration is written $H \rhd F$ and the transition relation between frame configurations is written $H_1 \rhd F_1 \rightarrow H_2 \rhd F_2$. A frame stack configuration is written $H \rhd FS$ and the transition relation between frame stack configurations is written $H_1 \rhd FS_1 \twoheadrightarrow H_2 \rhd FS_2$. Finally, a process configuration is written $H \rhd P$ and the transition relation between process configurations is written $H_1 \rhd P_1 \rightsquigarrow H_2 \rhd P_2$.

Simple frame transition rules:

$$H \rhd \langle L, x\texttt{=}c\texttt{;}\overline{s} \rangle^\ell \rightarrow H \rhd \langle L[x \mapsto c], \overline{s} \rangle^\ell \qquad \text{[E-Constant]}$$

$$H \rhd \langle L, x\texttt{=}y\texttt{;}\overline{s} \rangle^\ell \rightarrow H \rhd \langle L[x \mapsto L(y)], \overline{s} \rangle^\ell \qquad \text{[E-Var]}$$

$$H \rhd \langle L, x\texttt{=}y \oplus z\texttt{;}\overline{s} \rangle^\ell \rightarrow H \rhd \langle L[x \mapsto L(y) \oplus L(z)], \overline{s} \rangle^\ell \qquad \text{[E-Op]}$$

$$H \rhd \langle L, x\texttt{=}y\texttt{.}f\texttt{;}\overline{s} \rangle^\ell \rightarrow H \rhd \langle L[x \mapsto FM(f)], \overline{s} \rangle^\ell \text{ where } H(L(y)) = \langle \rho, FM \rangle \qquad \text{[E-Field]}$$

$$H \rhd \langle L, \texttt{if (}x\texttt{) \{}\overline{s}\texttt{\} else \{}\overline{t}\texttt{\}}\ \overline{u} \rangle^\ell \rightarrow H \rhd \langle L, \overline{s}\ \overline{u} \rangle^\ell \text{ where } L(x) = \texttt{true} \qquad \text{[E-CondEq]}$$
$$H \rhd \langle L, \texttt{if (}x\texttt{) \{}\overline{s}\texttt{\} else \{}\overline{t}\texttt{\}}\ \overline{u} \rangle^\ell \rightarrow H \rhd \langle L, \overline{t}\ \overline{u} \rangle^\ell \text{ where } L(x) = \texttt{false}$$

$$H \rhd \langle L, \texttt{while (}x\texttt{)\{}\overline{s}\texttt{\}}\ \overline{t} \rangle^\ell \rightarrow H \rhd \langle L, \overline{s}\ \texttt{while (}x\texttt{)\{}\overline{s}\texttt{\}}\ \overline{t} \rangle^\ell \text{ where } L(x) = \texttt{true} \qquad \text{[E-While]}$$
$$H \rhd \langle L, \texttt{while (}x\texttt{)\{}\overline{s}\texttt{\}}\ \overline{t} \rangle^\ell \rightarrow H \rhd \langle L, \overline{t} \rangle^\ell \text{ where } L(x) = \texttt{false}$$

$$H_0 \rhd \langle L, x\texttt{.}f\texttt{=}y\texttt{;}\overline{s} \rangle^\ell \rightarrow H_1 \rhd \langle L, \overline{s} \rangle^\ell \qquad \text{[E-Asn]}$$
where $L(x) = o, H_0(o) = \langle \sigma, FM \rangle$ and $H_1 = H_0[o \mapsto \langle \sigma, FM[f \mapsto L(y)] \rangle]$

$$H_0 \rhd \langle L, x\texttt{=new C();}\overline{s} \rangle^\ell \rightarrow H_1 \rhd \langle L[x \mapsto o], \overline{s} \rangle^\ell \qquad \text{[E-New]}$$
where $\mathit{fields}(\texttt{C}) = \overline{\tau}\ \overline{f}, o \notin \mathit{dom}(H_0)$ and $H_1 = H_0[o \mapsto \langle \texttt{C}, \overline{f} \mapsto \mathit{default}(\overline{\tau}) \rangle]$

In these transition rules the frames are labeled with meta-variable ℓ: they apply for both synchronous and asynchronous frames, factoring common semantics. Our transition rules $H_0 \rhd \langle L_0, \overline{s}\rangle^{\ell} \rightarrow H_1 \rhd \langle L_1, \overline{t}\rangle^{\ell}$ always preserve labels, i.e., a synchronous frame transitions to another synchronous frame, and an asynchronous frame transitions to an asynchronous frame with the *same* task. In rule [E-New] we use an auxiliary function, *default* that returns a default constant for a given type. This notion is taken from full C$^\sharp$ [12, §5.2] but for FC$^\sharp_5$ it simply maps type `int` to the value 0, type `bool` to the value `false` and all other types to the `null` literal. These simple transition rules are quite standard and for space reasons we do not elaborate on them further.

Next we consider the evaluation of a synchronous method call and returning from a synchronous method. For a `return` (though not a call) the label on the frame is important; as we shall see, the `return` rule is different for asynchronous frames.

Synchronous method call/return transition rules:

$H_0 \rhd F_0 \circ FS \twoheadrightarrow H_1 \rhd F_1 \circ FS \qquad \text{if } H_0 \rhd F_0 \rightarrow H_1 \rhd F_1$ [E-Frame]

$H \rhd \langle L_0, y_0{=}y_1\,.\,m(\overline{z})\,;\overline{s}\rangle^{\ell} \circ FS \twoheadrightarrow H \rhd \langle L_1, \overline{t}\rangle^{\mathsf{s}} \circ \langle L_0, \overline{s}\rangle^{\ell}_{y_0} \circ FS$ [E-Method-Exp]
where $H(L_0(y_1)) = \langle \rho, FM\rangle$, $mbody(\rho, m) = mb\colon (\overline{\sigma}\ \overline{x}) \rightarrow^{\mathsf{s}} \sigma_1$, $mb = \overline{\tau}\ \overline{y}\,;\ \overline{t}$ and
$L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \mathtt{this} \mapsto L_0(y_1)]$

$H \rhd \langle L_0, x\,.\,m(\overline{y})\,;\overline{s}\rangle^{\ell} \circ FS \twoheadrightarrow H \rhd \langle L_1, \overline{t}\rangle^{\mathsf{s}} \circ \langle L_0, \overline{s}\rangle^{\ell} \circ FS$ [E-Method-Stmt]
where $H(L_0(x)) = \langle \rho, FM\rangle$, $mbody(\rho, m) = mb\colon (\overline{\sigma}\ \overline{x}) \rightarrow^{\mathsf{s}} \mathtt{void}$, $mb = \overline{\tau}\ \overline{z}\,;\ \overline{t}$ and
$L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{z} \mapsto default(\overline{\tau}), \mathtt{this} \mapsto L_0(x)]$

$H \rhd \langle L_0, \mathtt{return}\ y\,;\overline{s}\rangle^{\mathsf{s}} \circ \langle L_1, \overline{t}\rangle^{\ell}_{x} \circ FS \twoheadrightarrow H \rhd \langle L_1[x \mapsto L_0(y)], \overline{t}\rangle^{\ell} \circ FS$ [E-Return-Val]

$H \rhd \langle L_0, \mathtt{return}\,;\overline{s}\rangle^{\mathsf{s}} \circ \langle L_1, \overline{t}\rangle^{\ell} \circ FS \twoheadrightarrow H \rhd \langle L_1, \overline{t}\rangle^{\ell} \circ FS$ [E-Return]

These transition rules are also quite standard. Rule [E-Frame] transitions the topmost, *active* frame of a frame stack. Rule [E-Method-Exp] transitions a method invocation. It first looks up in the heap the runtime type of the receiver. We make use of another auxiliary function induced by correct program: $mbody$ is a map from a type and a method name to a method body and an annotated type signature. For example, we write $mbody(\mathtt{C}, m) = mb\colon (\overline{\sigma}\ \overline{x}) \rightarrow^{\mathsf{s}} \phi$, when the method m in class `C` is a synchronous method, with formal parameters $\overline{\sigma}\ \overline{x}$, return type ϕ, and method body mb.

Rule [E-Method-Exp] applies when the receiver object supports method m and m is a synchronous method. In this case, we push a new synchronous frame (labeled `s`) on to the frame stack to execute the method body. Notice that we annotate the caller frame with the identifier that is waiting for the return value (this will be used in rule [E-Return-Val]). Rule [E-Method-Stmt] is similar except that m is a `void`-returning method, returning control. Note that the semantics of synchronous calls are the same whether issued from a synchronous or asynchronous frame (ℓ can be any label).

Rule [E-Return-Val] shows how a synchronous method returns a value to its caller. The caller frame, $\langle L_1, \overline{t}\rangle^{\ell}_{x}$, is waiting for a value for local identifier x. The active synchronous frame is popped and the caller frame becomes active and assigns the return value to x. Rule [E-Return] is similar except that no value is returned and the caller frame is not annotated with an identifier: the caller only expects control, not a value.

Asynchronous method call/return transition rules:

$H_0 \rhd \langle L_0, y_0{=}y_1\,.\,m(\overline{z})\,;\overline{s}\rangle^{\ell} \circ FS$ [E-Async-Method]
$\twoheadrightarrow H_1 \rhd \langle L_1, \overline{t}\rangle^{\mathsf{a}(o)} \circ \langle L_0[y_0 \mapsto o], \overline{s}\rangle^{\ell} \circ FS$
where $H_0(L_0(y_1)) = \langle \rho, FM\rangle,\ mbody(\rho, m) = mb\colon (\overline{\sigma}\ \overline{x}) \to^{\mathsf{a}} \psi,$ and $mb = \overline{\tau}\ \overline{y};\ \overline{t}$
$o \notin dom(H_0), H_1 = H_0[o \mapsto \langle \psi, \mathsf{running}(\epsilon)\rangle]$
$L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \mathtt{this} \mapsto L_0(y_1)]$

$H_0 \rhd \{\langle L, \mathtt{return}\ y\,;\overline{s}\rangle^{\mathsf{a}(o)} \circ FS\} \cup P$ [E-Async-Return]
$\rightsquigarrow H_1 \rhd \{FS\} \cup resume(\overline{F}) \cup P$
where $H_0(o) = \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(\overline{F})\rangle$ and $H_1 = H_0[o \mapsto \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{done}(L(y))\rangle]$

$H \rhd \langle L, x{=}\mathtt{await}\ y\,;\overline{s}\rangle^{\mathsf{a}(o)} \circ FS \twoheadrightarrow H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\mathsf{a}(o)} \circ FS$ [E-Await-Continue]
where $H(L(y)) = \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{done}(v)\rangle$

$H_0 \rhd \langle L, x{=}\mathtt{await}\ y\,;\overline{s}\rangle^{\mathsf{a}(o)} \circ FS \twoheadrightarrow H_1 \rhd FS$ [E-Await]
where $L(y) = o_1, H_0(o_1) = \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(\overline{F})\rangle$
$H_1 = H_0[o_1 \mapsto \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(\langle L, x{=}y\,.\,\mathtt{GetResult}()\,;\overline{s}\rangle^{\mathsf{a}(o)}, \overline{F})\rangle]$

$H \rhd \langle L, x{=}y\,.\,\mathtt{Result}()\,;\overline{s}\rangle^{\ell} \to H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\ell}$ [E-Result]
where $H(L(y)) = \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{done}(v)\rangle$

$H \rhd \langle L, x{=}y\,.\,\mathtt{Result}()\,;\overline{s}\rangle^{\ell} \to H \rhd \langle L, x{=}y\,.\,\mathtt{Result}()\,;\overline{s}\rangle^{\ell}$ [E-Result-Block]
where $H(L(y)) = \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(\overline{F})\rangle$

$H \rhd \langle L, x{=}y\,.\,\mathtt{GetResult}()\,;\overline{s}\rangle^{\ell} \to H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\ell}$ [E-GetResult]
where $H(L(y)) = \langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{done}(v)\rangle$

$H_0 \rhd \{\langle L, x{=}\mathtt{Task.AsyncIO{<}}\gamma\mathtt{>}()\,;\overline{s}\rangle^{\ell} \circ FS\} \cup P$ [E-Async-IO]
$\rightsquigarrow H_1 \rhd \{\langle L[x \mapsto o], \overline{s}\rangle^{\ell} \circ FS\} \cup P \cup \{\langle \{y \mapsto v\}, \mathtt{return}\ y\,;\rangle^{\mathsf{a}(o)} \circ \epsilon\}$
where $o \notin dom(H_0), H_1 = H_0[o \mapsto \langle \mathtt{Task{<}}\gamma\mathtt{>}, \mathsf{running}(\epsilon)\rangle]$ and $v \in Values(\gamma)$

These transition rules cover the new asynchronous features in C♯ 5.0. First we recall that task heap objects are of the form $\langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{done}(v)\rangle$ for some value v, or $\langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(\overline{F})\rangle$ where $\overline{F}$ is a sequence of frames—we will refer to this sequence as the *running state* of the task heap object. (In reality these two forms are encoded as conventional objects using delegates for the frames.) Tasks are *stateful*: a task heap object is created in initial state $\langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(\epsilon)\rangle$, with no waiters; can transition from state $\langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(\overline{F})\rangle$ to $\langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{running}(F_o, \overline{F})\rangle$, adding one waiter, and may terminate in a *completed* state $\langle \mathtt{Task{<}}\sigma\mathtt{>}, \mathsf{done}(v)\rangle$ for some value v of type σ. Once completed, a task cannot change state again.

Rule [E-Async-Method] shows how to transition a call to an asynchronous method. We create a fresh `Task` object in the heap (at address o), and set its state to be running. Initially, there are no waiters for this task, so its running sequence is empty. We push a new frame containing the method body on the frame stack and label it as asynchronous, i.e., with the label $\mathsf{a}(o)$. The caller frame is updated with the heap address of the task in its locals stack. Notice that the calling frame is *not* awaiting a value, just control.

[E-Async-Return] pops the active asynchronous frame, storing the return value in the task. It also resumes any waiters (there may be zero or more). The operation

$resume(\overline{F})$ is used to resume a sequence of suspended frames. It creates a bag of singleton frame stacks and is defined as $resume(\overline{F}) \stackrel{\text{def}}{=} \{\langle L, \overline{s}\rangle^{\ell} \circ \epsilon \mid \langle L, \overline{s}\rangle^{\ell} \in \overline{F}\}$.

Rule [E-Await-Continue] covers the case when a task being awaited is already completed. In this case we simply read out the value from the task and continue. Rule [E-Await] covers the case when the task being awaited is still running. In this case we need to pause the asynchronous method. Thus we pop the active asynchronous frame from the frame stack and add it to the sequence of awaiters of the incomplete task. Notice that we unfold the **`await`** y to y.`GetResult`()—once resumed, the first thing the frame will do is read the value from y's (completed) task object in the shared heap.

Rule [E-Result] and [E-Result-Block] implement the in-built method `Result` on tasks. If the task is completed then it returns the result; if it is running then it 'blocks' (which for simplicity we simulate by spinning, i.e. by transitioning to itself). In contrast, rule [E-GetResult] implements `GetResult`. It too returns the result if the task is completed. However, if the task is incomplete, no rule applies and the configuration is stuck (the implementation raises an exception). `GetResult` is *non-blocking* and *partial*.

Rule [E-Async-IO] models a prototypical asynchronous method. It immediately returns a fresh, running task to be completed, with some value v, by a separate thread.

Process transition rules:

$$H \rhd \{\epsilon\} \cup P \rightsquigarrow H \rhd P \qquad \text{[E-Exit]}$$

$$H_0 \rhd \{FS_0\} \cup P \rightsquigarrow H_1 \rhd \{FS_1\} \cup P \quad \text{if } H_0 \rhd FS_0 \twoheadrightarrow H_1 \rhd FS_1 \text{ [E-Schedule]}$$

Recall that a process is a collection of frame stacks, i.e., threads. Rule [E-Exit] deletes an empty frame stack from the process. Rather than formalizing a particular scheduler, rule [E-Schedule] simply transitions a process by non-deterministically selecting and transitioning a thread, possibly side-effecting the shared heap. Our semantics is an interleaved semantics, allowing preemption at every atomic statement.

4 Correctness Properties

Given our formalization of $\mathsf{FC}_5^\sharp$ we are able to prove some important correctness properties; specifically, type soundness. Interestingly, establishing these properties involves non-trivial extensions of the conventional techniques [4]. In this section we give some details of these extensions and the precise forms of the correctness properties; complete details are given in a technical report.

The typical approach to proving type soundness involves extending the notion of type checking to configurations, and then establishing preservation and progress properties. However, for $\mathsf{FC}_5^\sharp$ this is not strong enough—in particular to establish progress—we have to consider not only type correctness but also crucial non-interference properties of tasks; both those being executed on framestacks and also those that are waiters on others tasks (and so are suspended in the heap). We also need to establish that the stateful protocol of tasks described in §3.1—that tasks begin in an empty running state, acquire waiters and then terminate in a done state (and never transition once in a done state)—is preserved too.

Rather than combine the typing and non-interference properties into a single relation, we keep them separate (at the expense of more verbose theorem statements). The rules

for non-interference for processes, framestacks, frames, heaps and heap objects are as follows.

Non-interference properties:

$$\text{[Proc-ok]}\ \frac{\vdash (H \rhd FS_0)\ \mathbf{ok} \quad \cdots \quad \vdash (H \rhd FS_n)\ \mathbf{ok} \quad \forall i \neq j \in \{0..n\}.taskIds(FS_i)\#taskIds(FS_j)}{\vdash (H \rhd \{FS_0, \ldots, FS_n\})\ \mathbf{ok}}$$

$$\text{[EmpFS-ok]}\ \frac{}{H \vdash \epsilon\ \mathbf{ok}} \qquad \text{[FS-ok]}\ \frac{H \vdash F\ \mathbf{ok} \quad H \vdash FS\ \mathbf{ok} \quad taskIds(F)\#taskIds(FS)}{H \vdash F \circ FS\ \mathbf{ok}}$$

$$\text{[SF-ok]}\ \frac{}{H \vdash F^{\mathsf{s}}\ \mathbf{ok}} \qquad \text{[CSF-ok]}\ \frac{H \vdash F^{\mathsf{s}}\ \mathbf{ok}}{H \vdash F^{\mathsf{s}}_x\ \mathbf{ok}}$$

$$\text{[AF-ok]}\ \frac{Running(H(o)) \quad \forall o_1 \in dom(H).o \notin runningIds(H(o_1))}{H \vdash F^{\mathsf{a}(o)}\ \mathbf{ok}}$$

$$\text{[H-ok]}\ \frac{\forall o \in dom(H).H \vdash H(o)\ \mathbf{ok} \quad \forall o_1 \neq o_2 \in dom(H).runningIds(H(o_1))\#runningIds(H(o_2))}{\vdash H\ \mathbf{ok}}$$

$$\text{[HO-ok]}\ \frac{}{H \vdash \langle \mathtt{C}, FM\rangle\ \mathbf{ok}} \qquad \text{[DTHO-ok]}\ \frac{}{H \vdash \langle \mathtt{Task<}\sigma\mathtt{>}, \mathsf{done}(v)\rangle\ \mathbf{ok}}$$

$$\text{[RTHO-ok]}\ \frac{\forall i \neq j \in \{0..n\}.taskIds(F_i)\#taskIds(F_j) \quad \forall i \in \{0..n\}.\forall o \in taskIds(F_i).Running(H(o))}{H \vdash \langle \mathtt{Task<}\sigma\mathtt{>}, \mathsf{running}(F_0, \ldots, F_n)\rangle\ \mathbf{ok}}$$

We use a function $taskIds(FS)$ which returns the task ids of a frame stack FS (i.e., the set of all object ids o found in asynchronous frame labels $\mathsf{a}(o)$ in the frame stack). We also overload this function over frames. We use a function $runningIds$ that returns the task ids of the running state of a given task heap object. The predicate $Running$ tests whether the state of a task heap object is currently running.

Rule [Proc-ok] ensures that the task ids in the frame stacks in a process are pairwise disjoint (we use the symbol # to denote disjointness). Rule [FS-ok] ensures that in a frame stack the task ids are all distinct. Rule [AF-ok] ensures that any task id in an asynchronous frame label is not included in the task ids of any running state in the heap. Rule [H-ok] ensures that for all the task heap objects in the heap, the task ids of the running states are disjoint. Rule [RTHO-ok] ensures that a given running task heap object has no duplicate task ids in its running state, and that all task ids in its running state refer to running (non-completed) tasks.

We also define a relation between heaps that preserves the typing of the heap objects and also enforce non-interference of any new running state whilst bounding the task ids of any new running state.

Definition 1 (Heap evolution). *Heap H_0 evolves to H_1 wrt a set of task ids S, written $H_0 \leq_S H_1$ if (i) $\forall o \in dom(H_1)$. if $o \notin dom(H_0)$ and $H_1(o_1) = \langle \psi, \mathsf{running}(\overline{F})\rangle$ then $\overline{F} = \epsilon$, and (ii) $\forall o \in dom(H_0)$. if $H_0(o) = \langle \mathtt{C}, FM_0\rangle$ then $H_1(o) = \langle \mathtt{C}, FM_1\rangle$, if $H_0(o) = \langle \psi, \mathsf{done}(v)\rangle$ then $H_1(o) = \langle \psi, \mathsf{done}(v)\rangle$, and if $H_0(o) = \langle \psi, \mathsf{running}(\overline{F_0})\rangle$ then $H_1(o) = \langle \psi, \mathsf{running}(\overline{F_1}, \overline{F_0})\rangle$, $taskIds(\overline{F_0})\#taskIds(\overline{F_1})$ and $taskIds(F_1) \subseteq S$.*

We also have typing relations for processes, framestacks, frames and heaps, written $\vdash (H \rhd P) : \star$, $H \vdash FS : \phi_0 \to \phi_1$, $H \vdash F : \phi_0 \to \phi_1$ and $\vdash H : \star$, respectively. Space prevents us from giving definitions of these relations, but they are routine.

Theorem 1 (Preservation). *If* $\vdash H_0 : \star$ *and* $\vdash H_0$ **ok** *then:*

1. *If* $\Gamma; H_0 \vdash F_0 : \phi_0 \to \phi_1$, $H_0 \vdash F_0$ **ok** *and* $H_0 \rhd F_0 \to H_1 \rhd F_1$ *then* $\vdash H_1 : \star$, $\vdash H_1$ **ok**, $\Gamma; H_1 \vdash F_1 : \phi_0 \to \phi_1$, $H_1 \vdash F_1$ **ok** *and* $\forall S. H_0 \leq_S H_1$.
2. *If* $H_0 \vdash FS_0 : \phi_0 \to \phi_1$, $H_0 \vdash FS_0$ **ok** *and* $H_0 \rhd FS_0 \twoheadrightarrow H_1 \rhd FS_1$ *then* $\vdash H_1 : \star$, $\vdash H_1$ **ok**, $H_1 \vdash FS_1 : \phi_0 \to \phi_1$, $H_1 \vdash FS_1$ **ok** *and* $H_0 \leq_{taskIds(FS_0)} H_1$.
3. *If* $\vdash (H_0 \rhd P_0) : \star$, $\vdash (H_0 \rhd P_0)$ **ok** *and* $H_0 \rhd P_0 \rightsquigarrow H_1 \rhd P_1$ *then* $\vdash H_1 : \star$, $\vdash H_1$ **ok**, $\vdash (H_1 \rhd P_1) : \star$ *and* $\vdash (H_1 \rhd P_1)$ **ok**.

Proof. Part (1) is proved by case analysis on $H_0 \rhd F_0 \to H_1 \rhd F_1$. Part (2) is proved by induction on the derivation of $H_0 \rhd FS_0 \twoheadrightarrow H_1 \rhd FS_1$ and part (1), and part (3) by induction on the derivation of $H_0 \rhd P_0 \rightsquigarrow H_1 \rhd P_1$ and part (2).

Theorem 2 (Progress). *If* $\vdash (H_0 \rhd P_0) : \star$ *and* $\vdash (H_0 \rhd P_0)$ **ok** *then*

1. $H_0 \rhd P_0 \rightsquigarrow H_1 \rhd P_1$, *for some* H_1, P_1; *or*
2. *for all* $FS \in P_0$, *one of the following holds:*
   (a) $FS = \langle L, \texttt{return}\ x\,; \overline{t} \rangle^{\mathsf{s}} \circ \epsilon$.
   (b) $FS = \langle L, y\texttt{=}x\,.\,m\,\texttt{(}\overline{z}\texttt{);}\overline{t} \rangle^{l} \circ FS'$, *or* $FS = \langle L, x\,.\,m\,\texttt{(}\overline{z}\texttt{);}\overline{t} \rangle^{l} \circ FS'$, *or* $FS = \langle L, y\texttt{=}x\,.\,f\,\texttt{;}\overline{t} \rangle^{l} \circ FS'$, *or* $FS = \langle L, x\,.\,f\texttt{=}y\,\texttt{;}\overline{t} \rangle^{l} \circ FS'$, *where* $L(x) = \texttt{null}$.
   (c) $FS = \langle L, y\texttt{=}\texttt{await}\ x\,\texttt{;}\overline{t} \rangle^{\mathsf{a}(o)} \circ FS'$, *where* $L(x) = \texttt{null}$.
   (d) $FS = \langle L, \epsilon \rangle^{l} \circ FS'$.
   (e) $FS = \langle L, y\texttt{=}x\,\texttt{.GetResult();}\overline{t} \rangle^{l} \circ FS'$, *where* $L(x) = o$ *and* $H(L(x)) = \langle \texttt{Task<}\tau\texttt{>}, \mathsf{running}(\overline{F}) \rangle$.

The progress theorem states a well-formed process can either transition or must entirely consist of stacks in terminal or stuck states (the latter includes the case $P_0 = \{\}$). Case 2a, a terminal state, can only arise from finishing a call to a program's non-`void` main method. Cases 2b–2c are familiar and new expected stuck states due to `null` references. Case 2d is excluded by applying C♯'s restriction that all control paths in a (non-`void`) method body contain a `return` statement [12, §8.1]. Note that an asynchronous `return` $x\,\texttt{;}\overline{t}$ is never stuck due to the enclosing frame's task being in an unexpected $\mathsf{done}(\_)$ state; this potential case is ruled out by $\vdash H_0 \rhd P_0$ **ok**. Interestingly, we could also rule out case 2e by simply excluding any occurrences of `GetResult` in the original program; although the formal details are beyond the scope of this paper. With this restriction, the only occurrences of `GetResult` arise from the [E-Await] transition. These frames are only resumed by the rule [E-Async-Return] which also transitions the state of the task object to $\mathsf{done}(v)$. We can also show a property that no transition rule changes the state of a task that is completed back to running. These two properties allow us to show that case 2e does not arise for `GetResult`-free source programs.

5 Extensions

5.1 Extension 1: Optimized, One-Shot Semantics

The semantics presented so far is idealized: when an asynchronous frame is suspended to await a task, rule [E-Await] appends a *copy* of the frame to the task's list of waiters.

At first glance, the act of copying the frame appears to require an expensive allocation of a *fresh* frame to store its contents. Notice, however, that frames are never duplicated: after copying the frame, [E-Await] pops the active frame, discarding it to proceed with its continuation, the calling stack FS. Since frames are used in a *linear* fashion, the expensive allocation on each suspend is entirely avoidable. The trick to avoiding repeated allocation is to allocate just one container for each asynchronous frame and destructively update its contents at each suspension of that frame.

The C♯ 5.0 implementation does just this, representing a suspended frame on the heap as a "stateful" *delegate* of type `Action`. Delegates [12, Chapter 15] are just closures, containing the address of some environment and the address of some static code taking the environment as a first argument. Both addresses are immutable. The state of the frame is therefore maintained, not directly in the closure, but in its environment. To achieve this, the environment itself has mutable fields that store the current values of the frame's locals, its associated task, and the current state of the finite state machine. All read and writes of locals in the original code are compiled to indirected operations on fields of the environment. The delegate's code pointer just contains the fixed code interpreting the frame's state machine.

In this section, we formalize a high-level abstraction of this implementation. Our formalization makes the more efficient, destructive update explicit without descending all the way to the low-level representation of closures used in the concrete implementation. To do so, we require a new reference type, the delegate type `Action`. In our semantics, if not in the actual implementation, the heap representation of an action is just an object whose mutable state is a frame, containing some locals and statements. The locals map contains the *current* values of local variables. The statements represent the frame's original body in *some state of unfolding*, i.e., the frame's current "program counter". This allows us to adequately represent a paused frame, without exposing the compilation details of its encoding as a C♯ 4.0 delegate with a fixed pointer to a mutable environment and static code. Making this change also paves the way for our formalization of the *awaitable pattern* in §5.2.

First, we must extend $\mathsf{FC}_5^\sharp$ with the `Action` type and syntax for invoking an action:

$\mathsf{FC}_5^\sharp$ additional types and statements:

$\rho ::= \ldots \mid$ `Action`	Delegate reference type
$s ::= \ldots \mid a$`();`	Action invocation statement

We also need to extend and adjust our run-time representations. `Action` is a new reference type so action values are just addresses of objects in the heap. An `Action` object, $\langle$`Action`$, F\rangle$, contains a (mutable) frame F, storing locals, statements and label of a suspended frame. We also need to modify tasks to track, not waiting frames (running$(\overline{F})$), but waiting *actions*, represented as a sequence of *addresses* (running$(\overline{o})$). Thus a running task will have representation $\langle$`Task<`σ`>`, running$(\overline{o})\rangle$; completed tasks remains the same. The form of an asynchronous label, placed on frames, is now a(o_1, o_2). The new label carries not one but *two* addresses: the address of the frame's task, o_1, as before, and a second address, o_2, of an action. The action stores the previous state of the frame; recording its address in the frame label indicates where to save the next state of the frame prior to suspending.

Completing a task will need to resume a list of actions, not frames, so we adapt the definition of $resume(\overline{o})$ to set up appropriate synchronous stubs, one per action in $\overline{o}$: $resume(\overline{o}) \stackrel{\text{def}}{=} \{\langle\{x \mapsto o_i\}, x()\texttt{;return};\rangle^{\texttt{s}} \circ \epsilon \mid o_i \in \overline{o}\}$.

Asynchronous method transition rules (One-shot semantics):

$H \rhd \langle L, x()\texttt{;}\overline{s}\rangle^{\ell} \circ FS \twoheadrightarrow H \rhd F \circ \langle L, \overline{s}\rangle^{\ell} \circ FS$ [E-Action-Invoke]
where $H(L(x)) = \langle \texttt{Action}, F\rangle$

$H_0 \rhd \langle L_0, y_0\texttt{=}y_1\texttt{.}m\texttt{(}\overline{z}\texttt{);}\overline{s}\rangle^{\ell} \circ FS$ [E-Async-MethodOS]
$\rightarrow H_1 \rhd \langle L_1, \overline{t}\rangle^{\texttt{a}(o_1,o_2)} \circ \langle L_0[y_0 \mapsto o_1], \overline{s}\rangle^{\ell} \circ FS$
where $H_0(L_0(y_1)) = \langle \sigma_0, FM\rangle$ and $mbody(\sigma_0, m) = mb\texttt{:}\ (\overline{\sigma}\ \overline{x}) \rightarrow^{\texttt{a}} \psi$
$mb = \overline{\tau}\ \overline{y}\texttt{;}\ \overline{t}$ and $o_1, o_2 \notin dom(H_0), o_1 \neq o_2$
$H_1 = H_0[o_1 \mapsto \langle \psi, \texttt{running}(\epsilon)\rangle, o_2 \mapsto \langle , \rangle\texttt{Action}\langle L_1, \overline{t}\rangle^{\texttt{a}(o_1,o_2)}]$
$L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \texttt{this} \mapsto L_0(y_1)]$

$H_0 \rhd \{\langle L, \texttt{return}\ y\texttt{;}\overline{s}\rangle^{\texttt{a}(o_1,o_2)} \circ FS\} \cup P$ [E-Async-ReturnOS]
$\rightsquigarrow H_1 \rhd \{FS\} \cup resume(\overline{o}) \cup P$
where $H_0(o_1) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{running}(\overline{o})\rangle$
$H_1 = H_0[\ o_1 \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \texttt{done}(L(y))\rangle, o_2 \mapsto \langle \texttt{Action}, \langle L, \overline{s}\rangle^{\texttt{a}(o_1,o_2)}\rangle]$

$H \rhd \langle L, x\texttt{=await}\ y\texttt{;}\overline{s}\rangle^{\texttt{a}(o_1,o_2)} \circ FS$ [E-Await-ContinueOS]
$\twoheadrightarrow H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\texttt{a}(o_1,o_2)} \circ FS$ where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{done}(v)\rangle$

$H_0 \rhd \langle L, x\texttt{=await}\ y\texttt{;}\overline{s}\rangle^{\texttt{a}(o_1,o_2)} \circ FS \twoheadrightarrow H_1 \rhd FS$ [E-AwaitOS]
where $L(y) = o_3, H_0(o_3) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{running}(\overline{o})\rangle$
$H_1 = H_0[\ o_3 \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \texttt{running}(o_2, \overline{o})\rangle,$
$o_2 \mapsto \langle \texttt{Action}, \langle L, x\texttt{= GetResult}(y)\texttt{;}\ \overline{s}\rangle^{\texttt{a}(o_1,o_2)}\rangle]$

Rule [E-Action-Invoke] formalizes the invocation of an action, similar to a method call. Notice that the entire frame, including label, is restored from the heap. In particular, an asynchronous frame will continue to signal completion through its task and have access to its action (for future suspension, if needed).

Rule [E-Async-MethodOS] is similar to [E-Async-Method] but it additionally allocates a new `Action`, storing the initial state of the asynchronous method. The address of the action, o_2, is recorded in the extended label of the pushed frame.

Rule [E-Async-ReturnOS] is similar to [E-Async-Return], completing the asynchronous frame's task. Though it is not necessary, we save the current locals and continuation of the **`return`**, $\overline{s}$, in the frame's `Action`. For this simple semantics, it should be possible to show that this action can never be invoked again.[6]

Rule [E-Await-ContinueOS] is almost identical to [E-Await-Continue], continuing execution of the current frame with the argument's result. The only difference is the extended label. There is no need to update the value of o_2 at this point. Rule [E-AwaitOS] is similar to [E-Await], but the suspend mechanism is different. This rule writes the frame's current state, locals and continuation, to its associated action, stored at address o_2, available from the frame's label. It then adds the address of that action to the

[6] When we add support for the awaitable pattern, the potential for abuse of the awaitable protocol, will mean that this property no longer generally holds.

incomplete task's list of waiters. Notice how the state of the action in the heap is destructively modified - there is no way to "go back" to a previous state of this frame.

Consider rule [E-Async-MethodOS]. It directly pushes a new asynchronous frame and assigns its task, o_1, to the caller's variable, y_0. An alternative formulation would be to push a synchronous stub that invokes the new action, o_2, and then returns the task, o_1, to the waiting caller. This would be less direct, but equivalent, and somewhat more faithful to the actual implementation. For example, the implementation of `CopyToManual` from §2.1 is essentially a stub method that, when called, invokes its internal delegate, `act()`, before returning its task.

At this point, the change to using mutable state to represent suspended frames is just an optimization. The reason is that user-code is never provided with access to a suspended frame, so the change in semantics cannot be observed.

5.2 Extension 2: The Awaitable Pattern

As detailed in §2, in C♯ 5.0 it is possible to await not just tasks, but values of any *awaitable* type. Our formalization has assumed that the only awaitable type is `Task<`σ`>`. In this section, we embrace the full awaitable pattern, replacing rule [C-Await] with:

New typing rule for awaitable expressions ([C-Awaitable])

$$\frac{\begin{array}{cc} mtype(\sigma_0, \texttt{GetAwaiter}) = () \to \sigma_1 & mtype(\sigma_1, \texttt{IsCompleted}) = () \to \texttt{bool} \\ mtype(\sigma_1, \texttt{OnCompleted}) = (\texttt{Action}) \to \texttt{void} & mtype(\sigma_1, \texttt{GetResult}) = () \to \sigma_2 \end{array}}{\Gamma, x{:}\,\sigma_0 \vdash \texttt{await}\ x{:}\,\sigma_2}$$

We simplify C♯ 5.0 and assume the property `IsCompleted` is an ordinary *method*; the distinction between methods and properties is entirely cosmetic so nothing is lost.

In the transition semantics **`await`** expressions can no longer transition atomically but must, instead, be evaluated in multiple steps. These steps commence with obtaining the argument's awaiter and proceed with calls to the awaiter's members, thus interleaving (potentially) user-defined code with the semantics of the **`await`** construct. Rule [C-Awaitable] statically ensures that these dynamic unfoldings are well-typed.

But first, we need to arrange that tasks are awaitable and implement the remaining requirements of the awaitable pattern. Our system already provides an appropriate `GetResult` for tasks; we are left with providing `GetAwaiter`,`IsCompleted` and `OnCompleted`, ascribed with the following types:

$$\begin{aligned} mtype(\texttt{Task<}\sigma\texttt{>}, \texttt{GetAwaiter}) &= () \to \texttt{Task<}\sigma\texttt{>} \\ mtype(\texttt{Task<}\sigma\texttt{>}, \texttt{IsCompleted}) &= () \to \texttt{bool} \\ mtype(\texttt{Task<}\sigma\texttt{>}, \texttt{OnCompleted}) &= (\texttt{Action}) \to \texttt{void} \\ mtype(\texttt{Task<}\sigma\texttt{>}, \texttt{GetResult}) &= () \to \sigma \end{aligned}$$

To avoid hard-wiring C♯ 5.0's generic `TaskAwaiter<`σ`>` type, we simplify the C♯ 5.0 design and assume that `Task<`σ`>` is self-sufficient and serves as its *own* awaiter type. Correspondingly, x`.GetAwaiter()`'s type is just the type of task x; its implementation, by rule [E-Task-GetAwaiter] below, just returns the receiver.

Additional transition rules for `Task`'s awaitable operations:

$H \rhd \langle L, x{=}y\texttt{.GetAwaiter();}\overline{s}\rangle^{\ell}$ [E-Task-GetAwaiter]
$\rightarrow H \rhd \langle L[x \mapsto L(y)], \overline{s}\rangle^{\ell}$ where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, FM\rangle$

$H \rhd \langle L, x{=}y\texttt{.IsCompleted();}\overline{s}\rangle^{\ell}$ [E-Task-IsCompleted]
$\rightarrow H \rhd \langle L[x \mapsto \texttt{true}], \overline{s}\rangle^{\ell}$ where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \mathsf{done}(v)\rangle$
$\rightarrow H \rhd \langle L[x \mapsto \texttt{false}], \overline{s}\rangle^{\ell}$ where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \mathsf{running}(\overline{o})\rangle$

$H_0 \rhd \langle L, x\texttt{.OnCompleted}(y)\texttt{;}\overline{s}\rangle^{\ell} \rightarrow H_1 \rhd \langle L, \overline{s}\rangle^{\ell}$ [E-Task-OnCompleted-Suspend]
where $L(x) = o_1$ and $H_0(o_1) = \langle \texttt{Task<}\sigma\texttt{>}, \mathsf{running}(\overline{o})\rangle$
$L(y) = o_2$ and $H_1 = H_0[o_1 \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \mathsf{running}(o_2, \overline{o})\rangle$

$H \rhd \{\langle L, x\texttt{.OnCompleted(}y\texttt{);}\overline{s}\rangle^{\ell} \circ FS\} \cup P$ [E-Task-OnCompleted-Resume]
$\rightsquigarrow H \rhd \{\langle L, \overline{s}\rangle^{\ell} \circ FS\} \cup \mathit{resume}(o) \cup P$
where $H(L(x)) = \langle \texttt{Task<}\sigma\texttt{>}, \mathsf{done}(v)\rangle$ and $L(y) = o$

Task's implementation of `IsCompleted()` tests the state field of the receiver, returning **`true`** if and only if it is `done(_)`. The implementation of `OnCompleted`(y) adds its callback y (an `Action`), to the receiver's list of waiters. If the task is already completed, the action cannot be stored and must, instead, be resumed in the process. The latter rule is required since there is a race between testing that a task `IsCompleted()`, finding it is **`false`**, and calling `OnCompleted`(y)—some other thread could intervene and complete the task *before* `OnCompleted`(y) executes.

We can now formalize the operational semantics of **`await`** on any awaitable. Because we need to interleave the execution of methods from the awaitable pattern—which take several transitions and could be user-defined—with the semantics of **`await`**, we need to introduce two additional, transient *control* statements that can only appear within asynchronous frames.

FC$^{\sharp}_{5}$ additional control statements:

$s ::= \ldots \mid$ `suspend;` $\mid$ `getcc(Action` a`){`$\overline{s}$`};` suspend & get-current-continuation statements

Though artificial, these statements have direct interpretations as intermediate steps of a compiler generated finite-state-machine.[7]

We define $\mathit{unfold}(x = \texttt{await}\ y; \overline{s})^{(z,b)}$ to be the syntactic unfolding of an **`await`** as a new sequence of statements using temporaries z and b (note a is bound):

$$\mathit{unfold}(x = \texttt{await}\ y; \overline{s})^{(z,b)} \stackrel{\text{def}}{=} \begin{cases} z = y\texttt{.GetAwaiter();} \\ b = z\texttt{.IsCompleted();} \\ \texttt{if (}b\texttt{) \{\} else} \\ \texttt{\{ getcc(Action }a\texttt{)\{}z\texttt{.OnCompleted(}a\texttt{);suspend;\};\}} \\ x = z\texttt{.GetResult();} \\ \overline{s} \end{cases}$$

The operation unfolds an **`await`** of an awaitable object y by first retrieving its awaiter z and setting b to determine if the awaiter is complete. If complete, the code falls through

[7] For example, in our hand-coded `CopyToManual` from §2.1, `suspend` corresponds to a **`return`**; from the `act` delegate that pauses execution (cases 1 and 2 of the switch); `getcc(Action` a`){`$\overline{s}$`};` corresponds to advancing the (shared) `state` variable to the next logical state (following `getcc(Action` a`){`$\overline{s}$`};`) and accessing the task's state machine (`act`).

the conditional. If incomplete, the code transfers the current continuation of the getcc statement to z (through a) and suspends. The continuation of both the `true` branch and the getcc statement is just x = z`.GetResult();`$\overline{s}$. It assigns the result of the awaiter to x, and proceeds with the original continuation $\overline{s}$ of the `await`.

Awaitable pattern, asynchronous method transition rules:

$$H_0 \rhd \langle L_0, x\texttt{=await } y\texttt{;}\overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \quad \text{[E-Awaitable]}$$
$$\to H_0 \vdash \langle L_1, \mathit{unfold}(x \texttt{ = await } y\texttt{;}\overline{s})^{(z,b)}\rangle^{\mathsf{a}(o_1,o_2)}$$
$$\text{where } L_1 = L_0[z \mapsto \texttt{null}, b \mapsto \texttt{false}] \text{ and } z, b \notin \mathit{dom}(L_0), z \neq b$$

$$H_0 \rhd \langle L, \texttt{getcc(Action } a\texttt{)\{}\overline{s}\texttt{\};}\overline{t}\rangle^{\mathsf{a}(o_1,o_2)} \to H_1 \rhd \langle L[a \mapsto o_2], \overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \quad \text{[E-GetCC]}$$
$$\text{where } a \notin \mathit{dom}(L) \text{ and } H_1 = H_0[o_2 \mapsto \langle \texttt{Action}, \langle L, \overline{t}\rangle^{\mathsf{a}(o_1,o_2)}\rangle]$$

$$H \rhd \langle L, \mathsf{suspend}\texttt{;}\overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \circ FS \twoheadrightarrow H \rhd FS \quad \text{[E-Suspend]}$$

Rule [E-Awaitable] unfolds its await expression using fresh temporaries, z and b. In rule [E-GetCC], `getcc(Action` a`){`$\overline{s}$`}` unfolds by first saving its continuation $\overline{t}$ in the frame's task, o_2, discarding it from the active frame, and then entering the body $\overline{s}$. When $\overline{s}$ is just z`.OnCompleted(`a`);suspend;`, as per rule [E-Awaitable], $\overline{s}$ will *transfer* the current continuation to the awaiter and suspend.

In rule [E-Suspend] the suspend control statement pauses the asynchronous frame. This is similar to a `return`, but the frame's task is not marked completed and remains in its current state. One might expect this state to be running(_) but it may not be, depending on the semantics of `OnCompleted`.

Although our semantics unfolds `await`s dynamically, it is possible to statically expand well-typed `await` expressions by a source-to-source translation, sketched here:

$$[x = \texttt{await } y\texttt{;}\overline{s_0}]^{\Gamma} \stackrel{\text{def}}{=} (\Gamma_1, \overline{s_2}) \text{ where } \mathit{mtype}(\Gamma(y), \texttt{GetAwaiter}) = () \to \sigma_1$$
$$(\Gamma_1, \overline{s_1}) = [\overline{s_0}]^{\Gamma, z:\sigma_1, b:\texttt{bool}}$$
$$\overline{s_2} = \mathit{unfold}(x \texttt{ = await } y\texttt{;}\overline{s_1})^{(z,b)}$$
$$b, z \notin \mathit{dom}(\Gamma), b \neq z$$
$$[s\texttt{;}\overline{s}]^{\Gamma} \stackrel{\text{def}}{=} \ldots$$

This translation must be type-directed (in order to determine awaiter types) and needs to produce a new context as well as the list of statements in order to properly account for generated variables. Notice, however, that it is finite and does not need to duplicate the input continuation $\overline{s_0}$, making it suitable for compile-time expansion.

`OnCompleted`'s *one-shot* Restriction, Explained. Once we add the awaitable pattern to the mix, the optimization described in §5.1 becomes a proper change to the semantics, with observable consequences. The culprit is the awaitable pattern's `OnCompleted(`a`)` method since it provides user-code with access to the *one-shot* continuation, a, of the frame, represented not as a pure value but as a stateful object. Recall that our informal description of the awaiter pattern stipulated that implementations of `OnCompleted` are required to invoke their action *at most once*. The reason why should now be clear. Invoking the action will resume the frame and potentially modify the action's state. In our semantics, the update would happen at the next suspension. In the real implementation,

the update would happen at the next write to some notionally local, but actually shared, variable of the frame. Two concurrent invocations of the same action have unpredictable behaviour: each would race to save its next, possibly different state in the same action. Frame execution depends on the shared heap, modified non-deterministically by rule [E-Schedule], so two invocations could very easily reach different states.

As it happens, when extended with the awaitable pattern, even the simpler, copying semantics cannot tolerate multiple invocations of a continuation, but the reason is more subtle. In the copying semantics, even a copy of a frame is inherently stateful because its label will contain a reference to the original frame's task. Allocated on the heap, this task is shared state: several invocations of the same continuation would race, with possibly different results, to complete the very same task on exit. Part of the semantics of tasks is that they should complete at most once. This invariant is violated by any abuse of one-shot continuations, as enabled by the awaitable pattern.

6 Related Work

The debate regarding how asynchronous software should be structured is both old and ongoing. Lauer and Needham [14] noted that the thread-based and event-based models are dual; a program written in one style can be transformed into a program written in the other style. Though this establishes that the two models are equivalent in expressive power, it does not resolve the question of which model is easier to use or reason about.

Ousterhout [17] famously stated that "threads are a bad idea (for most programs)." His argument revolves around the claim that threads are more difficult to program than events because the programmer must reason about shared state, locks, race conditions, etc., and that they are only necessary when true concurrency–in contrast to asynchrony—is desired. Though he conflates the threaded model of programming, in which there is no inversion of control, with concurrency, his observation that the programmer should be able to reason about the operation of code is well-taken.

SEDA [25] demonstrates that the event model can be highly scalable. Servers designed using SEDA are broken into separate stages with associated event queues. Each stage automatically tunes its resource usage and computation to meet application-wide performance goals. Within each stage multiple threads may process events, but these threads are utilized only for concurrency. The programmer still has to manually manage the state associated with each event. SEDA's goal is to provide a self-tuning architecture that adapts to overload conditions, not to make programming servers easier.

The dual argument in favor of threads over events is made by von Behren et al. [22]. They tease apart the different aspects of threads that may make them undesirable and argue that most of these deficiencies are merely implementation artifacts. Capriccio [23] demonstrates that this is the case by providing a very efficient cooperative threading mechanism that avoids inversion of control and provides an efficient runtime. Because it uses cooperative threading, Capriccio avoids the overhead of concurrency, and code transformations to insert stack checks allow threads' stacks to grow without requiring large amounts of pre-allocated stack space. Like Capriccio, asynchronous C$^\sharp$ allows programs to be written in a natural way while providing an efficient implementation. However, instead of attempting to provide a general cooperative threading mechanism, it permits programmers to write asynchronous code in a straight-line fashion by

automating stack-ripping [1] via compilation to a state machine. Like the state machine translation C♯ employs, it is reminiscent of simpler coroutine implementations [21,7] built on Duff's device [6] that, however, make no attempt to maintain local state.

The observation that continuations provide a natural substrate on which to build a threading mechanism was made by Wand [24]. The observation that more restrictive, but more efficient, one-shot continuations suffice for continuation based threading dates back to [5]. Li and Zdancewic [15] use continuations to unify the event and thread-based models. They leverage the continuation monad in Haskell to allow programmers to write straight line code that is desugared into continuation passing style (CPS), thus allowing it to be used in an event-based IO framework they construct.

The *computation expressions* [20] of F♯ provide a generalized monadic syntax, which is essentially an extended form of Haskell's do-notation. When specialised to F♯'s *asynchronous workflow* monad—itself a continuation monad—they allow programmers to write monadic code that is syntactically expanded to explicit continuation-passing-code. This offers much of the legibility of programming in direct-style while, at the same time, providing access to the implicit continuations (as F♯ functions) whenever required (e.g., when supplying callbacks to asynchronous calls). The generality of computation expressions has a cost: each continuation of a monadic let is a heap-allocated function; and every wait on an asynchronous value typically requires an expensive allocation of a fresh continuation. This is similar to our idealized semantics in §3. The upshot is that these continuations can, in principle, be invoked several times, allowing the encoding of a much wider range of control operators than the one-shot actions of C♯'s feature. But there are more differences. In F♯, computation expressions produce *inert* values that are easily composed but must be explicitly run to produce a result. In C♯, on the other hand, each task returned by an async method call represents a *running* computation. This makes it easier to initiate asynchronous work but, perhaps, harder to define combinators that compose asynchronous methods. Though inspired by F♯, C♯ 5.0 support for asynchrony is quite different in performance, expressivity and usage.

Scala actors [11] provide an asynchronous, message passing model for programming with concurrent processes. Asynchronous programs may be written in terms of *receive*, which suspends the current thread until a message is received, or *react*, which enqueues a continuation that is called when a message is received; *receive* provides a thread-based interface and *react* provides an event-based interface to an underlying message passing framework. Although the two programming models are made similar through the use of various combinators, the programmer must still significantly modify code to move between styles. Rompf et al. [19] use a type and effect system to selectively CPS convert Scala programs, providing a less onerous path from threads to event-based asynchronous code, but C♯ 5.0's `async` keyword is even more lightweight.

Despite Ousterhout's early admonition that reasoning about threads is difficult and error-prone, none of the work mentioned makes an explicit attempt to provide programmers with a set of reasoning principles for asynchronous code. Although we believe C♯'s support for asynchrony exists at a useful point in the design space, our focus is on providing these reasoning principles. Threads or events, manual stack ripping or CPS, a programmer must have clear ways to reason about code behavior in order to build correct systems of any kind.

7 Conclusions

Real-world software construction demands effective methods for dealing with asynchrony. For such a method to be termed "effective," it must not require large-scale, manual code transformations such as stack ripping. Sequential computations should be expressible with sequential code, even if individual operations may execute asynchronously. Splitting sequential code up into a series of callbacks or explicitly rewriting it as a state machine is a steep price to pay, making code difficult to write, difficult to read, and difficult to reason about; if in doubt, contrast the synchronous, **`async`**-enabled, and hand-written state machine versions of the stream copying function from §2.1. While previous work has provided syntactic and library support for dealing with asynchrony, C♯ 5.0 brings this support to a widely-deployed, mainstream language.

One deficiency of this previous work is a lack of reasoning principles for asynchronous code. Our primary contribution is an operational semantics for C♯ 5.0 that allows programmers to answer questions about the code they write and make conclusions about the impact of adding asynchrony to their code. For example, using our semantics, the programmer can see that calling an **`async`** method does not spawn a new thread, but instead executes the method on the current stack. With the optimized semantics in §5.1, one can even begin to reason about space usage by, e.g., observing that the state of an **`async`** method is always stored in the same `Action`, allocated just once.

We plan to continue our formalization of C♯ 5.0 by incorporating additional language features, such as cancellation tokens and synchronization context object—we have already formalized exceptions and their interaction with **`async`**, but due to space restrictions this formalization, as well as an asynchronous tail-call optimization, is only available in a separate tech report.Our semantics have been translated to Coq. We will use this as a foundation to validate a translation from Featherweight C♯ 5.0—including the **`async`** construct—to Featherweight CIL, an idealized version of the bytecode targeted by the C♯ 5.0 compiler. Validation of this translation will prove that programmers can reason in terms of our high-level operational semantics even though the high-level program has been translated to bytecode and it is the bytecode that is actually executed.

While syntax is important for easing the pain of writing asynchronous code, a corresponding semantics is vital for writing correct software. With our semantics, C♯ 5.0 both provides relief and the necessary tools for thinking carefully about the remedy.

Acknowledgements. We thank the C♯ and Visual Basic teams for their collaboration, especially Lucian Wischik who led much of the design and implementation effort.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: Proceedings of USENIX (2002)
2. Bierman, G., Meijer, E., Torgersen, M.: Lost in translation: Formalizing proposed extensions to C♯. In: Proceedings of OOPSLA (2007)
3. Bierman, G., Meijer, E., Torgersen, M.: Adding Dynamic Types to C♯. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 76–100. Springer, Heidelberg (2010)
4. Bierman, G., Parkinson, M., Pitts, A.: MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory (2003)

5. Bruggeman, C., Waddell, O., Dybvig, R.K.: Representing control in the presence of one-shot continuations. In: Proceedings of PLDI (1996)
6. Duff, T.: Re: Explanation, please! (August 1988), USENET Article
7. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: Simplifying Event-Driven programming of Memory-Constrained embedded systems. In: Proceedings of SenSys (2006)
8. Fischer, J., Majumdar, R., Millstein, T.: Tasks: language support for event-driven programming. In: Proceedings of PEPM (2007)
9. Flanagan, C., Sabry, A., Duba, B., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of PLDI (1993)
10. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. Technical Report TR-97-293, Rice University (1997)
11. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
12. Hejlsberg, A., Torgersen, M., Wiltamuth, S., Golde, P.: The C♯ Programming Language, 4th edn. Addison-Wesley (2011)
13. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM TOPLAS 23(3), 396–450 (2001)
14. Lauer, H.C., Needham, R.M.: On the duality of operating system structures. Operating Systems Review 13(2), 3–19 (1979)
15. Li, P., Zdancewic, S.: Combining events and threads for scalable network services. In: Proceedings of PLDI (2007)
16. Microsoft Corporation. C# language specification for asynchronous functions (2011), http://msdn.microsoft.com/en-us/vstudio/async
17. Ousterhout, J.K.: Why threads are a bad idea (for most purposes). In: USENIX Winter Technical Conference, Invited Talk (June 1996)
18. Pierce, B., Turner, D.: Local type inference. In: Proceedings of POPL (1998)
19. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In: Proceedings of ICFP (2009)
20. Syme, D., Petricek, T., Lomov, D.: The F# Asynchronous Programming Model. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 175–189. Springer, Heidelberg (2011)
21. Tatham, S.: Coroutines in C (2000), http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
22. von Behren, R., Condit, J., Brewer, E.: Why events are a bad idea (for high-concurrency servers). In: Proceedings of HotOS (2003)
23. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: scalable threads for internet services. In: Proceedings of SOSP (2003)
24. Wand, M.: Continuation-based multiprocessing. In: Proceedings of LISP and Functional Programming (1980)
25. Welsh, M., Culler, D., Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services. In: Proceedings of SOSP (2001)

Lightweight Polymorphic Effects

Lukas Rytz, Martin Odersky, and Philipp Haller

EPFL, Switzerland
first.last@epfl.ch

Abstract. Type-and-effect systems are a well-studied approach for reasoning about the computational behavior of programs. Nevertheless, there is only one example of an effect system that has been adopted in a wide-spread industrial language: Java's checked exceptions. We believe that the main obstacle to using effect systems in day-to-day programming is their verbosity, especially when writing functions that are polymorphic in the effect of their argument. To overcome this issue, we propose a new syntactically lightweight technique for writing effect-polymorphic functions. We show its independence from a specific kind of side-effect by embedding it into a generic and extensible framework for checking effects of multiple domains. Finally, we verify the expressiveness and practicality of the system by implementing it for the Scala programming language.

1 Introduction

Type-and-effect systems are a well understood and widely used approach in the research literature for reasoning about computational effects. Originally designed to delimit the scope of dynamically allocated memory [21], the technique has been applied to various kinds of effects such as exceptions [8], purity [18], atomicity [1] or parallel programming [2]. Marino et al. [14] factor out the commonalities of different effect systems into a generic framework.

However, when taking a look at the most wide-spread programming languages used in industry, there is only one example of an effect system that has been put into practice: Java's checked exceptions. In addition, this particular system has earned a lot of critique about its verbosity and lack of expressiveness ([10], [22]), which in turn influenced language designers not to put effect systems into their languages ([10], [16]).

We believe that the fundamental property that makes effect systems expressive enough to be useful in everyday programming is effect-polymorphism. Functions are often implemented using delegation, by calling functions they receive as argument, and therefore the effect of a function can depend on the effect of its arguments. Polymorphic effect systems have been around for more than 20 years [13], and within limits,[1] Java also supports methods that are polymorphic in the thrown exception type.

[1] It is only possible to abstract over a fixed number of exception types

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 258–282, 2012.

```
abstract class List<T> {
 public <U> List<U> mapM(Function<T, U> f) throws Exception
 public <U, E extends Exception> List<U> mapP(FunctionE<T, U, E> f) throws E
}
```

This example shows two higher-order methods `mapM` and `mapP` in Java. The monomorphic version `mapM` can only accept functions with arbitrary effects as argument if it declares the effect `throws Exception`. The second version is polymorphic in the exception type of its argument function. Note that also the function type `FunctionE` has to be extended with an explicit exception type parameter.

The crucial issue is that writing effect-polymorphic methods results in code which is syntactically heavy and hard to understand. Also, polymorphism is tied to one specific effect domain: adding a new kind of effect system to Java would often also require to integrate a new syntax for effect-polymorphism.

In this paper, we propose a pragmatic and expressive new way for writing effect-polymorphic code. We present a system for lightweight polymorphic effect checking with the following contributions:

- We propose a syntax for writing effect-polymorphic functions which is as lightweight as writing an ordinary, monomorphic function, without the need for explicit effect parameters.
- To support these functions in the type system, we introduce a new kind of function type for effect-polymorphic functions. These types co-exist with ordinary function types. In a monomorphic function type $T \overset{e}{\Rightarrow} U$, the latent effect, the effect that might occur when the function is invoked, is annotated as e. For an effect-polymorphic function type $T \xrightarrow{e} U$, the latent effect consists of both e and the latent effect of its argument type T.
- We embed the new kind of functions into a generic effect checking framework which is independent of a specific effect domain. This extends the generic effect system proposed in [14] with effect-polymorphism. The framework is extensible and allows checking multiple kinds of effects at the same time, given a description of each effect domain. We show that every effect domain profits from the effect-polymorphism available through the framework.
- The described framework is implemented as a compiler plugin for the Scala programming language. We added effect checking for exceptions and successfully applied polymorphic effects to the core of the Scala collections library.

The rest of this paper is structured as follows. Section 2 gives an informal overview of the system, Section 3 presents the formalization, in Section 4 we report our practical experience, Section 5 discusses related work and Section 6 concludes.

2 Overview

In this section, we give an informal overview of our polymorphic type-and-effect system and we show that it constitutes a pragmatic and practical compromise between syntactic simplicity and expressivity.

2.1 Effect-Polymorphic Function Types

The main idea of our type-and-effect system is to introduce a new kind of function type which, by definition, denotes effect-polymorphic functions. A function is said to be effect-polymorphic if its *latent* effect, the effect that occurs when the function is applied, depends on the effect of its argument. To give an example, we define a simple higher-order function `hof` which applies its argument function to the constant `1`.

```
val hof = (f: Int ⇒ Int) → f 1
```

Intuitively, the effect of applying `hof` to a function `f` depends on the effect of the argument `f`. For instance, if "`f = (x: Int) ⇒ x + 1`" is a pure function,[2] then the invocation of `hof` does not have any side-effect. But if we apply it to a function "`f = (x: Int) ⇒ throw ex`" that throws an exception, then so does the invocation. We therefore conclude that the function `hof` is effect-polymorphic in its argument function `f`.

To differentiate between effect-polymorphic and ordinary, effect-monomorphic functions, we use two kinds of arrows in function types. The double arrow ⇒ is used for ordinary function types. For instance, the function `(x: Int) ⇒ x + 1` has type `Int` $\overset{\bot}{\Rightarrow}$ `Int`. The effect annotation on the arrow denotes the latent effect of the function, the effect that may occur when the function is applied. The symbol ⊥ denotes purity. If the effect annotation is omitted, the largest possible effect ⊤ is assumed.

An effect-polymorphic function type, such as the type of `hof`, is expressed with a single arrow →. Like ordinary function types, also polymorphic function types are annotated with an effect, however the default effect when the annotation is omitted is ⊥. So the function `hof` has the following type:

```
hof: (Int ⇒ Int) → Int // equivalent to (Int ⇒ Int) → Int
```

The crucial property of an effect-polymorphic function type is that its latent effect consist of two components:

- The annotated effect, an effect that may occur when the function is invoked, independently of the argument. In the example of `hof`, this effect is ⊥.
- The effect of the argument.

The second component is the source of effect-polymorphism. For each invocation of the polymorphic function, the effect of the argument can be different, which results in a different overall effect. We take a closer look at the types of the two function literals from the previous example:

```
val f: (Int ⇒ Int) = (x: Int) ⇒ x + 1
val g: (Int ⇒ Int) = (x: Int) ⇒ throw ex
```

[2] Instead of using the traditional abstraction syntax $\lambda x : T.t$, we write function literals in the form $(x : T) \Rightarrow t$

For each invocation of the function `hof`, the effect gets computed based on the actual argument type. Therefore, the invocation "`hof f`" has no effect, while the invocation "`hof g`" has the effect of throwing an exception.

2.2 Programming with Effect-Polymorphic Functions

We believe that the introduction of effect-polymorphic function types is a very efficient technique for adding polymorphic effect checking to a programming language, while keeping the annotation overhead for programmers within reasonable limits. To illustrate this point, we look at the higher-order function `map` which applies a given function to all elements of a list.

```
val map: IntList ⇒^⊥ (Int ⇒ Int) → IntList =
  (l: IntList) ⇒ (f: Int ⇒ Int) → l match {
    case Nil       => Nil
    case Cons x xs => Cons (f x) (map xs f)
  }
```

Even though the signature of the `map` function is fully effect-polymorphic, there is only one single effect annotation $\perp$, which denotes purity of the outer function. Since pure functions are very common, especially when currying is used to encode functions with multiple arguments, it might be worthwhile to introduce syntactic sugar for pure functions. In this case, the polymorphic `map` function would not need any effect annotation at all.

There are many higher-order functions that can be made effect-polymorphic by replacing a normal function type with an effect-polymorphic one. However, the type system we introduce does not only apply to functional programming. On the contrary: the presented ideas are based on our work on a polymorphic effect system for the Scala programming language. We decided to present the system using a simpler lambda calculus in order not to distract from the main concepts. In Section 4 we show how lightweight effect-polymorphism can be expressed in the object-oriented setting, and we present the results obtained with our implementation for Scala in Section 4.3.

Effect-polymorphism is a crucial ingredient to make effect-checking practicable in a real-world programming language, and this applies equally to object-oriented languages. For instance, many of the object-oriented design patterns identified in [5] are based on delegation. In order to correctly annotate the effect of methods which are implemented by calling methods of their parameters, polymorphic effect annotations are indispensable. The widely used strategy pattern is the most prominent example: it basically models higher-order functions. A method that is implemented in terms of its argument strategy is polymorphic in the effect of that strategy.

The effect system for checked exceptions in Java illustrates the need for effect-polymorphism. It is possible in Java to write methods that are polymorphic in the thrown exception type, but doing so is very verbose and therefore often avoided in practice. This limitation is at the source of the well-known issues

with Java's checked exceptions ([15], [22]): `throws` declarations are often copy-pasted from the callee to the caller method, which has a negative impact on the maintainability and readability of the program code.

2.3 An Extensible Framework for Multiple Effect Domains

The polymorphic type-and-effect system outlined in the previous section is not tied to a specific kind of side effect, such as exceptions that might be thrown or state that might be modified. Instead, it defines an extensible framework that allows effect checking of multiple effect domains in the same language, at the same time.

In order to add a new kind of side effect to be checked by the framework, a description of the effect domain in the form of a semi-lattice has to be provided. The semi-lattice consists of a set of effects for the domain, a join operation to compute the combination of two effects, and a sub-effect relation which compares two effects.[3] The following example shows a simple effect lattice for tracking IO effects:

- Effect set: $E_{\mathcal{I}} = \{noIO, IO\}$
- Join operation: $e_1 \sqcup_{\mathcal{I}} e_2 = \begin{cases} IO & \text{if } (e_1 = IO) \vee (e_2 = IO) \\ noIO & \text{otherwise} \end{cases}$
- Sub-effect relation: $e_1 \sqsubseteq_{\mathcal{I}} e_2 = (e_1 = noIO) \vee (e_2 = IO)$

The framework also needs to know the top and bottom elements of each effect lattice. In the case of IO effects, these are $\top_{\mathcal{I}} = IO$ and $\bot_{\mathcal{I}} = noIO$.

In addition to the effect lattice, every concrete effect domain has to define the effect associated with each syntactic construct of the language. For instance, an effect system for tracking exceptions declares that a `throw` expression adds an effect, and that a `try` expression can mask effects. This information has to be provided in the form of a function $\mathit{eff}_{\mathcal{D}}$ which receives as argument a representation of the program fragment in question and returns its side-effect.[4] This function is closely related to the "adjust" function in [14], and we will give a precise definition in Section 3.3.

In the domain of IO, effects are introduced by calling pre-defined functions that have a latent IO effect. There are no syntactic constructs that introduce or mask IO effects, therefore the $\mathit{eff}_{\mathcal{I}}$ function can be left unspecified and the framework will use a default definition.

Annotating Multiple Effect Domains. Since the effect checking framework supports tracking effects from multiple domains, the effect annotations on function types have to declare an effect for every domain that is being checked. This is achieved by annotating the function types with a tuple consisting of

[3] Note that the lattice operations need to fulfill the common lattice properties, such as transitivity for $\sqsubseteq$. Also, the join and sub-effect operations are related: for all effects e_1, e_2, we have $e_1 \sqsubseteq e_2 \iff e_1 \sqcup e_2 = e_2$.

[4] We will discuss the effect domain of exceptions in detail in Section 3.4.

domain-specific effect annotations. For instance, if there are three effect domains $\mathcal{D}1$, $\mathcal{D}2$ and $\mathcal{D}3$, a function type has the form $T_1 \xRightarrow{e_{\mathcal{D}1}\ e_{\mathcal{D}2}\ e_{\mathcal{D}3}} T_2$.

However, this example shows that the annotation scheme does not scale very well: effect annotations quickly become long and are hard to maintain. When adding a new effect domain, the annotation in every function type needs to be updated. Fortunately, there is a simple solution to this problem. It is based on the observation that in many cases, function types have either no effect, a small number of effects, or the topmost effect. To backup this claim, we look at a few examples:

- The function `map` accepts as argument a function that can have any effect. Therefore, this argument is annotated with the topmost effect.
- The implementation of `map` itself is pure, there is no other effect than the effect of its argument.
- The `map` function is only one example of a large class of library functions that are pure. The same is true for instance for all operations on immutable datatypes.
- Functions that do have side-effects usually have effects in one or very few effect-domains. For instance, a random generator is non-deterministic, operations on mutable data structures modify state, or functions from a file-API have IO-effects. In addition, these functions might have exceptional behavior. However, it is rather uncommon to have functions with side-effects from all those domains at the same time.

In order to simplify the multi-domain effect annotations and take the above observations into account, we introduce two specific effect annotations which, by definition, range over all effect domains: $\top$ and $\bot$. When used as such, the two annotations have the expected meaning: $\top = \top_{\mathcal{D}1}\ \ldots\ \top_{\mathcal{D}N}$, similarly for $\bot$.

The crucial characteristic however is that the multi-domain annotations can be combined with concrete effect annotations from individual effect domains. For instance, the type $T_1 \xRightarrow{\bot\ e_{\mathcal{D}i}} T_2$ denotes a function which can have effect $e_{\mathcal{D}i}$ in the domain $\mathcal{D}i$, but is pure in all other domains. Similarly, combining the $\top$ annotation

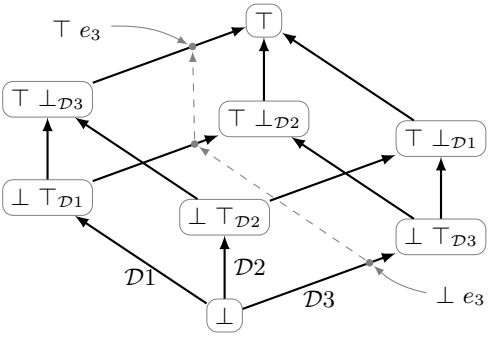


Fig. 1. Effect annotations in multiple domains

with concrete effects restricts the allowed effect in certain domains. This behavior is illustrated in Figure 1, showing an example with three effect domains.

3 Formalization

In order to formalize the ideas presented in the previous section, we extend a simply typed lambda calculus with effect annotations and effect-polymorphic function types. The syntax of the formal language is summarized in Figure 2. Note that the syntax for function abstraction is different than usual and does not use the λ symbol.

t	$::=$	x	parameter
		$\mid t\ t$	application
		$\mid v$	value
v	$::=$	$(x : T) \Rightarrow t$	monomorphic abstraction
		$\mid (x : T) \rightarrow t$	effect-polymorphic abstraction
T	$::=$	$T \overset{e}{\Rightarrow} T$	function type
		$\mid T \overset{e}{\rightarrow} T$	effect-polymorphic function type
e	$::=$	$\bot\ e_D \mid \top\ e_D \mid e_D$	effect annotation
e_D	$::=$	$e_D\ e_D \mid \cdot$	concrete effects
Γ	$::=$	$\emptyset \mid \Gamma, x : T$	parameter context
f	$::=$	$\epsilon \mid x$	polymorphism context

Fig. 2. Core language syntax

The effect annotation e on a function type declares the *latent* effect, the effect that may occur when the function is invoked. Note that e defines an effect for every active effect domain. Subsection 3.1 explains how the integration of multiple effect domains into one effect system is handled.

There are two kinds of functions: ordinary, monomorphic functions denoted using the double arrow $\Rightarrow$, and effect-polymorphic functions denoted with a single arrow $\rightarrow$. The two kinds of arrows appear in function abstraction terms and in function types.

A monomorphic function type $T_1 \overset{e}{\Rightarrow} T_2$ declares a latent effect e, the effect that may occur when the function is applied. In the type of an effect-polymorphic function $T_1 \overset{e}{\rightarrow} T_2$ however, the annotated effect e does not denote the entire latent effect of the function. Instead, the effect of such a function consists of two parts: the concrete, annotated effect e and the effect of its argument of type T_1. Only higher-order functions, functions that take another function as argument, can be effect-polymorphic. This invariant is checked by the typing rule T-Abs-Poly which enforces the parameter type T_1 to be a function type. The effect of the argument function is implicitly added to the total effect of an effect-polymorphic function.

For syntactic convenience, the effect annotations on function types can be omitted, in which case the following default effects are used:

- $T_1 \Rightarrow T_2$ is a equivalent to $T_1 \overset{\top}{\Rightarrow} T_2$
- $T_1 \to T_2$ is a equivalent to $T_1 \overset{\bot}{\longrightarrow} T_2$

Example 1. We inspect the type of the simple higher-order function `hof` introduced in Section 1:

```
val hof: (Int ⇒[⊤] Int) →[⊥] Int = (f: Int ⇒[⊤] Int) → f 1
```

Using the default effects mentioned above, the effect annotations in the type of `hof` as well as the one in the function abstraction can be omitted:

```
val hof: (Int ⇒ Int) → Int = (f: Int ⇒ Int) → f 1
```

3.1 A Multi-domain Effect Lattice

When checking multiple kinds of effects at the same time, every effect domain needs to be described as a join-semilattice as explained in Section 2.3. For a domain $\mathcal{D}$, the lattice consists of a set of atomic effects $E_{\mathcal{D}}$, a join operation $\sqcup_{\mathcal{D}}$ and a sub-effect relation $\sqsubseteq_{\mathcal{D}}$. Additionally, the bottom and top elements $\bot_{\mathcal{D}}$ and $\top_{\mathcal{D}}$ of $E_{\mathcal{D}}$ have to be specified.

These individual domains are combined into one multi-domain effect lattice that is used in this section. The elements of this lattice are tuples of effects from the individual domains:

$$E = \{e_{\mathcal{D}1} \dots e_{\mathcal{D}n} \mid e_{\mathcal{D}1} \in E_{\mathcal{D}1} \wedge \dots \wedge e_{\mathcal{D}n} \in E_{\mathcal{D}n}\}$$

The $\sqcup$ and $\sqsubseteq$ operations are defined element-wise using $\sqcup_{\mathcal{D}i}$ and $\sqsubseteq_{\mathcal{D}i}$ for every domain $\mathcal{D}i$. We omit their definitions here for brevity.

3.2 Subtyping

The subtyping relation of our calculus has the common reflexivity and transitivity properties.

$$\frac{}{T <: T}\ (\text{S-Refl}) \qquad \frac{T' <: S \qquad S <: T}{T' <: T}\ (\text{S-Trans})$$

The subtyping rules covering the two kinds of function types in our system are entirely symmetrical.

$$\frac{T_1 <: T_1' \qquad T_2' <: T_2 \qquad e' \sqsubseteq e}{T_1' \overset{e'}{\Rightarrow} T_2' <: T_1 \overset{e}{\Rightarrow} T_2}\ (\text{S-Fun-Mono}) \qquad \frac{T_1 <: T_1' \qquad T_2' <: T_2 \qquad e' \sqsubseteq e}{T_1' \overset{e'}{\longrightarrow} T_2' <: T_1 \overset{e}{\longrightarrow} T_2}\ (\text{S-Fun-Poly})$$

In S-Fun-Mono, a function with a latent effect e' can only be a subtype of another function with effect e if $e' \sqsubseteq e$. As an example, we take a higher-order function `hof` that requires its argument to be pure:

```
val pureHof = (f: Int ⇒[⊥] Int) ⇒ f 1
```

The subtyping rule will only allow pure functions to be passed into `pureHof`.

When looking at effect-polymorphic function types in S-Fun-Poly, remember that we defined previously the latent effect of $T_1 \xrightarrow{e} T_2$ to consists of two parts: the annotated effect e plus the latent effect of the argument type T_1. This raises the question why the subtyping rule for polymorphic function types only compares the annotated effects. Assume we have two functions:

```
val maybePure: (Int ⇒[⊤] Int) →[⊥] Int = ...
val pure: (Int ⇒[⊥] Int) →[⊥] Int = ...
```

In general, an invocation of `maybePure` might have any effect, while an invocation of `pure` is always pure. However, the subtyping relation seems to contradict this observation: due to contra-variance of arguments, the type of `maybePure` is a subtype of the type of `pure`.

To build an intuition why the subtyping rule is correct, we take a closer look at the two function types. The type of `pure` says: "Give me a pure function from `Int` to `Int`, and I compute a result without producing a side-effect." For instance, in the body of a function `m`

```
val m = (pure: (Int ⇒[⊥] Int) →[⊥] Int) ⇒ ...
```

the function `pure` *only* accepts pure functions. Now assume that we use the function `maybePure` where a function of the type of `pure` is expected, e.g.

```
m maybePure
```

As explained before, this is allowed by the subtyping rules. We can now see that it is also correct, because in the body of method `m` only pure functions will be passed into `maybePure`. Due to effect-polymorphism, those invocations of `maybePure` have no effect. In other words, the type of the function `maybePure` says: "If you give me a pure function, I *also* compute a result without producing a side-effect!"

3.3 Static Semantics

Extensible Type-and-Effect Checking. Since we are creating an extensible framework for tracking side-effects of multiple effect domains, we want to give each concrete effect system the possibility of customizing the effect of evaluating a term. For that reason, the typing rules introduced in this section are parametrized by an auxiliary function *eff*.

For every effect domain $\mathcal{D}$, the function $\mathit{eff}_{\mathcal{D}}$ computes the effect of evaluating a term, given the effects of its sub-terms. It takes two arguments: a name indicating the syntactic form in question, and a list of effects of its sub-terms. By default it combines all the argument effects using the $\sqcup_{\mathcal{D}}$ operator:

$$\mathit{eff}_{\mathcal{D}}(*, \overline{e}) = \bigsqcup_{\mathcal{D}} \overline{e}$$

The default $\mathit{eff}_{\mathcal{D}}$ function can be specialized by concrete effect domains; we will discuss the example of exceptions in Section 3.4. However, arbitrary $\mathit{eff}_{\mathcal{D}}$

functions can make the type-and-effect system unsound. For the system to be correct, the $\mathit{eff}_{\mathcal{D}}$ functions needs to meet the following monotonicity requirement.

Lemma 1. *Monotonicity.*
For every effect domain $\mathcal{D}$ and every syntactic form Trm,
1. *if* $\forall\ e_i \in e, d_i \in d\ .\ e_i \sqsubseteq d_i$, *then* $\mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, \overline{e}) \sqsubseteq \mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, \overline{d})$
2. $\mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, e_1, \ldots, e_{i1} \sqcup e_{i2}, \ldots, e_n) \sqsubseteq \mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, e_1, \ldots, e_{i1}, \ldots, e_n) \sqcup e_{i2}$

The first clause of the monotonicity lemma requires the $\mathit{eff}_{\mathcal{D}}$ functions to be monotonic. Implementing effect masking remains possible, as we will see in the effect domain of exceptions. The second part of the monotonicity lemma prevents the output effect to depend on the presence of a certain input effect. This restriction falls in line with the general semantics of effect annotations in type-and-effect systems: an annotated effect *may* occur, but it is not required to occur.

The function eff used in the typing statements works on all effect domains at the same time, similar to the multi-domain lattice operations described in Section 3.1. It is composed of the individual $\mathit{eff}_{\mathcal{D}i}$ functions in the straightforward way:

$$\mathit{eff}(\textsc{Trm}, \overline{e}) = \mathit{eff}_{\mathcal{D}1}(\textsc{Trm}, \overline{e})\ \ldots\ \mathit{eff}_{\mathcal{D}n}(\textsc{Trm}, \overline{e})$$

Typing Rules. Terms are assigned a type and an effect using a judgement of the form $\Gamma; f \vdash t : T\ !\ e$ where Γ maps variables to their types. The additional environment variable f is used for type-checking effect-polymorphic methods. While its exact role will be discussed later, remember for now that it holds either a parameter $x \in \Gamma$, or the special value ϵ which is distinct from all parameter names.

As is common, referencing a parameter does not have a side-effect:

$$\frac{x : T \in \Gamma}{\Gamma; f \vdash x : T\ !\ \bot} \qquad \text{(T-Param)}$$

Next, we look at the typing rules for monomorphic function abstraction and application.

$$\frac{\Gamma, x : T_1; \epsilon \vdash t : T_2\ !\ e}{\Gamma; f \vdash (x : T_1) \Rightarrow t : T_1 \xRightarrow{e} T_2\ !\ \bot} \qquad \text{(T-Abs-Mono)}$$

$$\frac{\Gamma; f \vdash t_1 : T_1 \xRightarrow{e} T\ !\ e_1 \quad \Gamma; f \vdash t_2 : T_2\ !\ e_2 \quad T_2 <: T_1}{\Gamma; f \vdash t_1\ t_2 : T\ !\ \mathit{eff}(\textsc{App}, e_1, e_2, e)} \qquad \text{(T-App-Mono)}$$

The typing rule for abstraction infers the result type T_2 and the latent effect e of a function. By using the value ϵ in the environment for type-checking the function body, we propagate the information that the term belongs to a monomorphic function.

The rule T-App-Mono is a standard typing rule for method applications. The resulting effect consists of three parts: e_1 is the effect of evaluating the function, e_2 is the effect of evaluating the argument and e is the latent effect of

the function. These three effects are combined using the *eff* function introduced in the beginning of this section.[5]

Next we analyze the typing rules for effect-polymorphic function abstractions and invocations. But before that, we take a close look at the functionality of the extended typing environment $\Gamma; f$.

Suppose we are analyzing the effect of a simple effect-polymorphic function

```
val hof = (f: Int ⇒^⊤ Int) → f 1
```

The computed type should be $(\texttt{Int} \overset{\top}{\Rightarrow} \texttt{Int}) \xrightarrow{\bot} \texttt{Int}$, i.e., the function `hof` itself has effect $\bot$. The effect of invoking `f` can be ignored because it is already expressed in the function type through effect-polymorphism.

To achieve this special treatment of the argument function, the parameter f is placed in the extended environment as $\Gamma; f$ when type-checking the function body of an effect-polymorphic function.

$$\frac{T_1 = T_a \overset{e_1}{\Rightarrow} T_b \qquad \Gamma, f : T_1; f \vdash t : T_2 \,!\, e}{\Gamma; f' \vdash (f : T_1) \to t : T_1 \overset{e}{\to} T_2 \,!\, \bot} \quad \text{(T-Abs-Poly)}$$

Note that the typing rule forces the argument type T_1 to be a monomorphic function type — only higher-order functions can be effect-polymorphic. We will explain later why the argument function has to be monomorphic.

The following typing rule T-App-Param implements the mentioned special treatment of the argument function f:

$$\frac{\begin{array}{c} f : T_1 \overset{e}{\Rightarrow} T \in \Gamma \\ \Gamma; f \vdash t : T_2 \,!\, e_2 \qquad T_2 <: T_1 \end{array}}{\Gamma; f \vdash f\ t : T \,!\, \mathit{eff}(\text{App}, \bot, e_2, \bot)} \quad \text{(T-App-Param)}$$

When applying a function f which is the parameter of an enclosing effect-polymorphic function, then the latent effect of f is not taken into account.

The last element of the static semantics is the typing rule for invocations of effect-polymorphic functions.

$$\frac{\begin{array}{c} \Gamma; f \vdash t_1 : T_1 \overset{e}{\to} T \,!\, e_1 \\ \Gamma; f \vdash t_2 : T_2 \,!\, e_2 \qquad T_2 <: T_1 \end{array}}{\Gamma; f \vdash t_1\ t_2 : T \,!\, \mathit{eff}(\text{App}, e_1, e_2, e \sqcup \mathit{latent}(T_2))} \quad \text{(T-App-Poly)}$$

There is one single but crucial difference to the rule T-App-Mono for monomorphic function applications. The latent effect of the function t_1 consists of two components: the concrete effect e annotated in the function type, and the latent effect of the argument function t_2 which is computed using $\mathit{latent}(T_2)$.

Note that the rule T-App-Poly is at the root of our effect-polymorphic type system. We obtain effect-polymorphism by computing for each invocation of t_1 the effect of the actual argument type T_2.

[5] Remember that by default, *eff* computes the join of its argument effects

The parameter type T_1 is known to be a monomorphic function type: this is enforced by the typing rule T-Abs-Poly. Since $T_2 <: T_1$, we know that also T_2 is a monomorphic function type. Therefore, the auxiliary function computing the latent effect is simply defined as

$$latent(T) = e \qquad \text{where } T = T_1 \xRightarrow{e} T_2$$

The reason why only monomorphic functions are allowed as parameters of effect-polymorphic functions is that the typing rules become simpler without decreasing the expressiveness. Imagine that a polymorphic function takes another polymorphic function as argument:

```
val highHof = (f: (Int ⇒ Int) → Int) → f ((x: Int) ⇒ x + 1)
```

When looking at the type of `highHof`, the information that its argument `f` is applied to a *pure* function cannot be recovered. Therefore, there is no advantage in allowing effect-polymorphic functions as arguments.

3.4 Examples of Concrete Effect Domains

We now present two extensions of the core calculus that implement effect checking for two concrete effect domains. Both extensions are orthogonal to the mechanisms of effect-polymorphism in the base language. Every concrete effect system that is added to the framework profits from effect-polymorphism without any additional effort: the extensions would be exactly the same in a monomorphic effect checking framework.

Exceptions. In order to add effect checking for exceptions, we first need to extend the base language with primitives to throw and handle exceptions. The additions to the language syntax are presented in Figure 3. We use a finite set of exceptions $p_1 \ldots p_n$ that can be thrown and caught, however the system could be easily extended to an open hierarchy of effects such as the exception types in languages like Scala or Java. An effect annotation `throws`$(\overline{p})$ denotes that any of the exceptions in $\overline{p}$ might be thrown. The effect lattice for the exception domain $\mathcal{E}$ is defined in Figure 4.

To give a valid type to the `throw` primitive, we introduce a bottom type `Nothing` which is a subtype of every other type.

t	::=	...	
		| `throw`(p)	throwing an exception
		| `try` t `catch`$(\overline{p})$ t	catching and handling exceptions
T	::=	...	
		| `Nothing`	bottom type
e_D	::=	...	
		| `throws`$(\overline{p})$	exception effect annotation
p	::=	$p_1 \mid \ldots \mid p_n$	exceptions

Fig. 3. Extended syntax for exceptions

$$E_{\mathcal{E}} = \{\texttt{throws}(p) \mid p \subseteq \{p_1, \ldots, p_i\}\} \qquad \texttt{throws}(\overline{p}) \sqsubseteq_{\mathcal{E}} \texttt{throws}(\overline{q}) \iff p \subseteq q$$
$$\bot_{\mathcal{E}} = \texttt{throws}() \qquad \texttt{throws}(\overline{p}) \sqcup_{\mathcal{E}} \texttt{throws}(\overline{q}) = \texttt{throws}(\overline{p} \cup \overline{q})$$
$$\top_{\mathcal{E}} = \texttt{throws}(p_i, ..., p_n)$$

Fig. 4. Effect lattice for exceptions

$$\frac{}{\texttt{Nothing} <: T} \qquad \text{(S-Nothing)}$$

The typing rules for the two new syntactic forms are defined as follows:

$$\frac{e = \mathit{eff}(\textsc{Throw}(p))}{\Gamma; f \vdash \texttt{throw}(p) : \texttt{Nothing} \,!\, e} \qquad \text{(T-Throw)}$$

$$\frac{\begin{array}{c}\Gamma; f \vdash t_1 : T_1 \,!\, e_1 \qquad T_1 <: T \\ \Gamma; f \vdash t_2 : T_2 \,!\, e_2 \qquad T_2 <: T \\ e_t = \mathit{eff}(\textsc{Try}, e_1) \qquad e = \mathit{eff}(\textsc{Catch}(\overline{p}), e_t, e_2)\end{array}}{\Gamma; f \vdash \texttt{try}\ t_1\ \texttt{catch}(\overline{p})\ t_2 : T \,!\, e} \qquad \text{(T-Try)}$$

Finally, to complete the description of the new effect domain we have to inform the framework that `throw` expressions can add effects, while `try` expressions can mask effects. This is achieved by defining the function $\mathit{eff}_{\mathcal{E}}$:

$$\begin{array}{lll}\mathit{eff}_{\mathcal{E}}(\textsc{Throw}(p)) & = \texttt{throws}(p) & \\ \mathit{eff}_{\mathcal{E}}(\textsc{Try}, e) & = e & \\ \mathit{eff}_{\mathcal{E}}(\textsc{Catch}(\overline{p}), e_1, e_2) & = \texttt{throws}((\overline{q} \setminus \overline{p}) \cup \overline{s}) & \text{where } \texttt{throws}(\overline{q}) \in e_1 \\ & & \texttt{throws}(\overline{s}) \in e_2\end{array}$$

Asynchronous Operations. The second extension that we present can be used to check the correct and/or efficient use of asynchronous computations. Several popular languages, including C#, F# [20], and Scala, support constructs to start a computation asynchronously, returning a handle (typically called a "future") used to retrieve the result, once it becomes available.

For example, in Scala a long-running computation can be started as follows:

```
val ft = future {
  // long-running computation
}
```

For efficiency, the body of `future` is executed on a thread pool. `ft` is a handle for the result; when retrieving its result, it blocks the current thread until the future's result has been computed or an unhandled exception has been thrown in the future's body:

```
val result = ft()
```

Retrieving the `result` as shown above is a blocking operation. However, when run using a fixed-size thread pool, calling blocking operations inside the bodies

t	::= ...	
	| `future` t	asynchronous expression
	| `block`	blocking expression
	| `blocking` t	delimiting blocking expression
e_D	::= ...	
	| `B` | `noB`	blocking / non-blocking effect annotation

Fig. 5. Extended syntax for asynchronous operations

of futures is problematic. Blocking operations are operations that may cause the underlying thread to wait indefinitely. Examples are waiting for the completion of a synchronous I/O operation, or waiting on a condition variable inside a monitor. Such thread-blocking operations may lead to starvation, and, in the worst case, may lock up the entire thread pool [9].

Using an effect system, it is possible to prevent these system-induced deadlocks at compile time. The idea is to require wrapping blocking operations, so that the underlying thread pool can be resized temporarily. This approach to supporting blocking operations has been adopted in the fork/join pool of Java 7 [12]. In the following we present an effect system which guarantees that all blocking operations are properly wrapped, thereby eliminating an entire class of concurrency errors when using thread pools.

The additions to the language syntax are presented in Figure 5. For simplicity, we use a fixed blocking expression `block`; in practice, many more expressions could be potentially blocking, for example, functions for synchronization or blocking I/O. The `future` t expression asynchronously runs an expression t which must be non-blocking, i.e., pure in this effect system. The fact that an expression is blocking is expressed using the `B` effect annotation. We omit the definition of the effect lattice, since it is trivial in this case. Finally, the `blocking` t expression wraps a potentially blocking expression t, such that the effect of the wrapped expression is non-blocking.

The typing rules for the three new syntactic forms are defined as follows:

$$\frac{\Gamma; \epsilon \vdash t : T \,!\, e \qquad \texttt{B} \notin e}{\Gamma; f \vdash \texttt{future}\ t : T \,!\, \mathit{eff}(\textsc{Future}, e)} \qquad \text{(T-Future)}$$

$$\frac{}{\Gamma; f \vdash \texttt{block} : T \,!\, \mathit{eff}(\textsc{Block})} \qquad \text{(T-Block)}$$

$$\frac{\Gamma; f \vdash t : T \,!\, e}{\Gamma; f \vdash \texttt{blocking}\ t : T \,!\, \mathit{eff}(\textsc{Blocking}, e)} \qquad \text{(T-Blocking)}$$

Finally, to complete the description of the new effect domain we have to define an $\mathit{eff}_{\mathcal{B}}$ function:

$$\begin{aligned} \mathit{eff}_{\mathcal{B}}(\textsc{Future}, e) &= e \\ \mathit{eff}_{\mathcal{B}}(\textsc{Block}) &= \texttt{B} \\ \mathit{eff}_{\mathcal{B}}(\textsc{Blocking}, e_1) &= \texttt{noB} \end{aligned}$$

The $\mathit{eff}_{\mathcal{B}}$ function expresses the fact that `block` expressions add a blocking effect, while `blocking` expressions mask a blocking effect.

3.5 Dynamic Semantics

In order to model the runtime behavior of our formal language we define a big-step operational semantics. A term t reduces in one step to either a value v or an error $\texttt{throw}(p)$, written $t \Downarrow \langle r, S\rangle$ where $r ::= v \mid \texttt{throw}(p)$. The set S contains the effects that occurred while evaluating the term. Every element $e \in S$ is an atomic effect, i.e., $S \subseteq E$ where E is the effect lattice defined in Section 3.1.

Extensible Effect Domains. In the sprit of extensibility to multiple effect domains, the evaluation rules are parametrized by an auxiliary function *dynEff* which computes the effect of evaluating a term based on the effects of its subterms. This function is closely related to the function *eff* used in the typing judgements, but it operates on sets of effects instead of atomic effects. The reason is that in contrast to the static semantics, the operational semantics does not approximate the occurrence of two distinct atomic effects by their join, but keeps both effects in the resulting set S.

In the case of exceptions, the function $\mathit{dynEff}_{\mathcal{E}}$ is defined as follows:

$$
\begin{aligned}
\mathit{dynEff}_{\mathcal{E}}(\textsc{App}, S_1, S_2, S_l) &= S_1 \cup S_2 \cup S_l \\
\mathit{dynEff}_{\mathcal{E}}(\textsc{Throw}(p)) &= \texttt{throws}(p) \\
\mathit{dynEff}_{\mathcal{E}}(\textsc{Try}, S) &= S \\
\mathit{dynEff}_{\mathcal{E}}(\textsc{Catch}(\overline{p}), S_1, S_2) &= (S_1 \setminus \{\texttt{throws}(p_i) \mid p_i \in \overline{p}\}) \cup S_2
\end{aligned}
$$

In order for the type system to be sound, the *eff* function needs to model *dynEff* conservatively and correctly. This requirement is explained in Section 3.6.

Evaluation Rules. We now present the evaluation rules.

$$
\frac{t_1 \Downarrow \langle \texttt{throw}(p), S_1\rangle \qquad S = \mathit{dynEff}(\textsc{App}, S_1, \emptyset, \emptyset)}{t_1\ t_2 \Downarrow \langle \texttt{throw}(p), S\rangle}\ (\text{E-App-E1})
$$

$$
\frac{t_1 \Downarrow \langle v_1, S_1\rangle \qquad t_2 \Downarrow \langle \texttt{throw}(p), S_2\rangle \qquad S = \mathit{dynEff}(\textsc{App}, S_1, S_2, \emptyset)}{t_1\ t_2 \Downarrow \langle \texttt{throw}(p), S\rangle}\ (\text{E-App-E2})
$$

When evaluating an application, if one of the two terms evaluates to $\texttt{throw}(p)$ for some exception p, then so does the entire expression.

$$
\frac{t_1 \Downarrow \langle (x : T) \mapsto t, S_1\rangle \qquad t_2 \Downarrow \langle v_2, S_2\rangle \qquad t[v_2/x] \Downarrow \langle r, S_l\rangle \qquad S = \mathit{dynEff}(\textsc{App}, S_1, S_2, S_l)}{t_1\ t_2 \Downarrow \langle r, S\rangle}\ (\text{E-App})
$$

In the evaluation rule for applications, we write $t[v/x]$ for the term t with all occurrences of the variable x replaced by value v. We use the special arrow $\mapsto$ to range over both, effect-polymorphic and monomorphic functions.

$$\frac{S = \mathit{dynEff}(\textsc{Throw}(p))}{\texttt{throw}(p) \Downarrow \langle \texttt{throw}(p), S\rangle} \quad \text{(E-Throw)}$$

A `throw` expression does not evaluate, but the evaluation rule still computes the set of dynamic effects of the expression.

$$\frac{\begin{array}{c} t_1 \Downarrow \langle \texttt{throw}(p), S_1\rangle \qquad p \in \overline{p} \\ t_2 \Downarrow \langle r_2, S_2\rangle \qquad S_t = \mathit{dynEff}(\textsc{Try}, S_1) \\ S = \mathit{dynEff}(\textsc{Catch}(\overline{p}), S_t, S_2) \end{array}}{\texttt{try}\ t_1\ \texttt{catch}(\overline{p})\ t_2 \Downarrow \langle r_2, S\rangle} \quad \text{(E-Try-E)}$$

The evaluation of a `try-catch` expression depends on the result obtained for the first subterm t_1. In case it evaluates to an error $\texttt{throw}(p)$, and the exception p is handled by the `catch` clause, then the final result is the evaluation of the handler t_2. Otherwise the following rule applies.

$$\frac{t_1 \Downarrow \langle r_1, S_1\rangle \qquad S = \mathit{dynEff}(\textsc{Try}, S_1)}{\texttt{try}\ t_1\ \texttt{catch}(\overline{p})\ t_2 \Downarrow \langle r_1, S\rangle} \quad \text{(E-Try)}$$

The last evaluation rule applies to `try-catch` expressions in which the evaluation of t_1 either does not raise an exception, or it raises an exception that is not handled by the $\texttt{catch}(\overline{p})$ clause. In this case, the obtained result r_1, which might be an error, is propagated.

3.6 Effect Soundness

In this section we state two important theorems for the soundness of the type system presented in Section 3.3 with respect to the dynamic semantics introduced in the previous section.

We use the following notational convenience: in the static semantics, every expression has an effect e, while in the dynamic semantics, the evaluation of an expression yields a *set* of effects S. We write $S \preceq e$ to express that every effect in S is smaller than e, i.e., $\forall e_s \in S\ .\ e_s \sqsubseteq e$.

Theorem 1. *Preservation.*
If $\Gamma; f \vdash t : T\ !\ e$ is a valid typing statement for term t and the term evaluates as $t \Downarrow \langle r, S\rangle$, then there is valid a typing statement $\Gamma; f \vdash r : T'\ !\ e'$ for r with $T' <: T$.

Theorem 2. *Effect soundness.*
If $\Gamma; f \vdash t : T\ !\ e$ and $t \Downarrow \langle r, S\rangle$, then $S \preceq e \sqcup \text{latent}(\Gamma(f))$.

The effect soundness theorem states that every effect that occurs when evaluating a term t is represented in the typing derivation for t. Remember that in the typing rule for effect-polymorphic functions, T-Abs-Poly, the argument function f is propagated in the extended environment $\Gamma; f$. Invocations of f are thereafter treated as pure by typing rule T-App-Param.

Therefore, given a typing statement $\Gamma; f \vdash t : T\ !\ e$, the effect that might occur when evaluating t consists of e *and* the latent effect of f, *latent*$(\Gamma(f))$.

Consistency Requirement. In both semantics, we use an auxiliary function to compute the effect that occurs when evaluating a term. The preservation and soundness theorems are based on the assumption that the static *eff* function conservatively models the behavior of the *dynEff* function in the operational semantics.

Lemma 2. *Consistency.*

- *Let* $S = \text{dynEff}(\text{TRM}, \overline{S})$ *be the set of dynamic effects that occur when evaluating a term* t *of the form* TRM. *The list* $\overline{S}$ *contains an effect set for every subterm of* t.
- *Let* $\Gamma; f$ *be an environment and* $\overline{e}$ *be a list of static effects such that every effect set in* $\overline{S}$ *is approximated by* $S_i \preceq e_i \sqcup \text{latent}(\Gamma(f))$.
- *Then the static effect* $e = \text{eff}(\text{TRM}, \overline{e})$ *is a conservative approximation of the effects in* S, *i.e.,* $S \preceq e \sqcup \text{latent}(\Gamma(f))$.

This consistency lemma has to be verified for every effect domain. In the case of exceptions or asynchronous operations, the verification is straightforward and therefore omitted here.

Proof Sketch. We give a proof sketch for the effect soundness theorem. In addition to preservation, the proof makes use of a lemma showing that effects are preserved in typing statements under value substitution. This lemma comes in two flavors: one for monomorphic and one for effect-polymorphic abstractions.

Lemma 3. *Preservation under substitution for monomorphic abstractions.*
If $\Gamma, x : T_1; f \vdash t : T \mathbin{!} e_l$, $f \neq x$ *and* $\Gamma; g \vdash v : T_2 \mathbin{!} \bot$ *with* $T_2 <: T_1$,
then $\Gamma; f \vdash t[v/x] : T' \mathbin{!} e'_l$ *such that* $T' <: T$ *and* $e'_l \sqsubseteq e_l$.

Lemma 4. *Preservation under substitution for polymorphic abstractions.*
If $\Gamma, x : T_1; x \vdash t : T \mathbin{!} e_l$ *and* $\Gamma; f \vdash v : T_2 \mathbin{!} \bot$ *with* $T_2 <: T_1$,
then $\Gamma; \epsilon \vdash t[v/x] : T' \mathbin{!} e'_l$ *such that* $T' <: T$ *and* $e'_l \sqsubseteq e_l \sqcup \text{latent}(T_2)$.

The two lemmas state that the type and the effect of a term t decrease when a free variable in t is replaced by a value with a conforming type.

The proof of Theorem 2 is carried out using induction on the evaluation rules for a term t. We look at the most interesting case E-APP that produces the following derivations:

$t = t_1\ t_2$
$t_1 \Downarrow \langle (x : T'_1) \mapsto t_{11}, S_1 \rangle$
$t_2 \Downarrow \langle v_2, S_2 \rangle$
$t_{11}[v_2/x] \Downarrow \langle r, S_l \rangle$
$S = \mathit{dynEff}(\text{APP}, S_1, S_2, S_l)$

There are multiple typing rules for type-checking an application expression. We investigate the key case T-APP-POLY and obtain the following sub-derivations:

$$\Gamma; f \vdash t_1 : T_1 \xrightarrow{e_l} T \,!\, e_1$$
$$\Gamma; f \vdash t_2 : T_2 \,!\, e_2$$
$$T_2 <: T_1$$
$$e = \mathit{eff}(\textsc{App}, e_1, e_2, e_l \sqcup \mathit{latent}(T_2))$$

Our goal is to show that in environment $\Gamma; f$, the static effect e correctly approximates the dynamic effects S, i.e., $S \preceq e \sqcup \mathit{latent}(\Gamma(f))$.

We see that t_1 evaluates to a function abstraction. The preservation theorem states that the type of this resulting function is a subtype of t_1's original type $T_1 \xrightarrow{e_l} T$. Since the term is a function abstraction, the subtyping rules restrict the type to be a polymorphic function type. Looking at the canonical forms, we observe that the term can only be a polymorphic function abstraction $(x : T_1') \to t_{11}$, and we obtain the following typing derivation:

$$\Gamma, x : T_1'; x \vdash t_{11} : T' \,!\, e_l' \qquad \text{with } T_1 <: T_1',\ T' <: T \text{ and } e_l' \sqsubseteq e_l$$

Applying preservation to the term t_2, we obtain $v_2 : T_2'$ with $T_2' <: T_2$. Using transitivity of subtyping, we obtain $T_2' <: T_1'$, and we can apply the substitution Lemma 4 to obtain

$$\Gamma; \epsilon \vdash t_{11}[v_2/x] : T'' \,!\, e_l'' \qquad \text{with } T'' <: T' \text{ and } e_l'' \sqsubseteq e_l' \sqcup \mathit{latent}(T_2')$$

By applying the induction hypothesis on the subterm $t_{11}[v_2/x]$, we obtain

$$S_l \preceq e_l'' \sqcup \mathit{latent}(\Gamma(\epsilon))$$
$$S_l \preceq e_l' \sqcup \mathit{latent}(T_2') \qquad \text{by } e_l'' \sqsubseteq e_l' \sqcup \mathit{latent}(T_2') \text{ and } \mathit{latent}(\Gamma(\epsilon)) = \bot$$

Since $T_2' <: T_2$ we can easily verify that $\mathit{latent}(T_2') \sqsubseteq \mathit{latent}(T_2)$. Together with the induction hypotheses on t_1 and t_2, we now have the necessary conditions to apply the consistency Lemma 2:

$$S_1 \preceq e_1 \sqcup \mathit{latent}(\Gamma(f))$$
$$S_2 \preceq e_2 \sqcup \mathit{latent}(\Gamma(f))$$
$$S_l \preceq e_l \sqcup \mathit{latent}(T_2)$$

We obtain the desired result:

$$\mathit{dynEff}(\textsc{App}, S_1, S_2, S_l) \preceq \mathit{eff}(\textsc{App}, e_1, e_2, e_l \sqcup \mathit{latent}(T_2)) \sqcup \mathit{latent}(\Gamma(f))$$

The proofs of the remaining cases are conducted in a similar fashion. A full proof for all lemmas and both theorems is available in a separate technical report [19].

4 Lightweight Polymorphic Effects in Scala

We implemented the ideas presented in the previous section as a generic framework for polymorphic effect-checking in the Scala programming language. The implementation comes in the form of a compiler plugin which adds new phases to the compilation pipeline. These phases are executed after the unaltered type-checking transformation and therefore, effect-checking can be seen as a pluggable type system [3].

4.1 Effect-Polymorphic Methods

Using the same ideas as in the formal system presented above, we extended Scala with a new syntactic form for defining effect-polymorphic methods. While ordinary methods are defined using the keyword `def`, an effect-polymorphic method is introduced with `fun`:[6]

```
fun h(f: Int => Int): Int = f(1)
```

The method `h` is polymorphic in the effect of its argument. When applying it to a pure function, then that invocation of `h` does not have a side-effect:

```
h(x => x + 1)
```

However, we glossed over one important property of Scala: it is an object-oriented language, and the arguments passed to methods are objects. This is equally true for first-class functions like `f` from the example. Concretely, a unary function in Scala is represented as an object of type `Function1`:

- The trait `Function1` has one abstract method named `apply`:
  ```
  trait Function1[+A, -R] {
    def apply(x: A): R
  }
  ```
- The type `Int => Int` is a shorthand for `Function1[Int, Int]`
- The function literal `x => x + 1` is syntactic sugar for an anonymous class[7]
  ```
  new Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
  }
  ```

This observation raises the question what it means for the method `h` to be effect-polymorphic in its argument `f`, since `f` is an object. The answer is that in the object-oriented case, a method is polymorphic in the effect of the *member methods* of its argument. In the example, method `h` is effect-polymorphic in the `apply` method of its argument `f`.

The definition of effect-polymorphism for the object-oriented case is a slight extension of the definition we used in the formal system: a method can be effect-polymorphic in multiple argument methods. For instance, in

```
fun m(a: A): B = a.b()
```

the method `m` is effect-polymorphic in all members of `a`, not only in `a.b`. In the same way, if an effect-polymorphic method has multiple parameters, it is effect-polymorphic in all of them.

This extension is however not a fundamental change, because methods cannot be partially applied. This means that when a method is invoked, all the parameters are defined and therefore all the argument methods and their effects are known. The same situation could be simulated in our calculus using tuples.

[6] This syntactic adjustment is the only language change that was performed. Alternatively, we could also have left the language unchanged and used an annotation.

[7] Local type inference defines the two type parameters of `Function1`

4.2 Effect Annotations in Scala

In order to annotate the latent effect of a method in Scala, our framework uses standard type annotations on the return type of the method. For instance, the following signature describes a method that might throw an exception:

```
def doIO(file: String): Unit @throws[IOException] = ...
```

Ordinary methods defined with the keyword `def` are impure by default. The method `doIO` therefore allows any side-effects in effect domains other than exceptions. The annotation `@pure` marks a method as pure in all effect domains, like the $\bot$ annotation in Section 3.1.

Effect-polymorphic methods defined using `fun` are pure by default.

One question is how the effect of anonymous functions should be annotated. For instance, the function literal `(x: Int) => x + 1` has type `Function1[Int, Int]`, but this type does not have any effect annotation.

In order to propagate the effect information of anonymous functions using their types, the effect checking framework makes use of refinement types [16]. Concretely, the type of the above function literal is extended to

```
Function1[Int, Int] {
  def apply(x: Int): Int @pure
}
```

To make function types with effects more compact, we are planning to introduce syntactic sugar for the above case, for instance `Int` $\Rightarrow$ `Int @pure`.

4.3 Practical Experience

We verified the expressiveness of our polymorphic effect system by applying it to the Scala collections framework. To illustrate the process of making a library effect-polymorphic, we look at the method `map` which applies an argument function to all elements of a collection. This method is implemented at the root of the collection hierarchy in class `TraversableLike` [17] and shared by all descending collection classes such as lists, maps, sets and vectors.

In essence, the parent collection class `TraversableLike` has one abstract method `foreach` which we make effect-polymorphic by changing `def` to `fun`:

```
fun foreach[U](f: Elem => U): Unit
```

Using this method, the class provides concrete implementations of common collection operations that are shared across all collection types: `filter`, `map`, `flatMap`, `partition`, `forall` and many more. The implementation of method `map` is as follows:

```
trait TraversableLike[+A, +Repr]
  // ...
  fun map[B, That](f: A => B,
                   implicit bf: CanBuildFrom[Repr, B, That]): That = {
    val b: Builder[B, That] = bf.apply(this)
    this.foreach(x => b += f(x))
```

```
    b.result
  }
}
```

A detailed explanation of this code can be found in [17]. For every collection type extending `TraversableLike`, the type parameter `A` represents the element type of the collection, and `Repr` is the collection type itself. The method `map` takes the target element type `B` and the target collection type `That` as argument. Allowing `map` to produce a different collection type than the current `Repr` is important in some situations, as explained in [17].

The additional value argument `bf` is called the "builder factory". It is used to obtain a builder object of type `Builder[B, That]`. The builder is a simple buffer which collects elements of type `B` and produces a collection of type `That` with these elements. Note that the builder factory is an implicit argument which does not have to be provided by the programmer when using `map`, instead it is searched and inserted by the compiler.

Using the method `foreach`, every element of the current collection is mapped with the argument function `f` and added to the builder. At the end, the resulting collection is obtained from the builder using `result`.

As previously with `foreach`, the only change we performed to make `map` effect-polymorphic is changing the definition keyword from `def` to `fun`. Now, the method `map` is effect-polymorphic in the argument function `f`, and in all member methods of the builder factory `bf`. However, the class `CanBuildFrom` has only one member method which is annotated pure:

```
trait CanBuildFrom[-From, -Elem, +To] {
  def apply(from: From): Builder[Elem, To] @pure
}
```

Therefore the builder factory does not contribute any effect, and `map` is effect-polymorphic in its argument function `f`.

Remember that effect-polymorphic functions are pure by default. Therefore, the implementation of `map` is not allowed to have any side-effects, except the effect of the argument `f`. If we look for instance at exceptions, purity can be easily verified:

- Invoking `bf.apply` does not have an effect, as already discussed
- Adding elements to the builder and obtaining the final result do not throw exceptions (i.e., the `+=` and `result` methods are pure)
- The effect of calling `foreach` is the effect of the argument function; the function `x => b += f(x)` calls the function `f`, which is allowed by effect-polymorphism

This line of reasoning is applied by the framework for every checked effect domain. Verifying purity with respect to state modifications in this example is less straightforward. However, a recent type-and-effect system for purity [18] is expressive enough to verify this case, and we are working on integrating it into our framework.

If an effect system is added to the framework in which the implementation of `map` is not pure, then the return type of `map` has to be annotated with the corresponding effect.

4.4 Implementing Concrete Effect Domains

We implemented the framework for polymorphic type-and-effect systems presented in this paper as a plugin for the Scala compiler. Extending the framework with a new effect domain is simple, as explained in Section 2.3: the programmer needs to provide

- an implementation of the effect lattice,
- the annotation definitions for annotating latent effects in the source code,
- two methods describing how to serialize and de-serialize the annotations into lattice elements, and
- an implementation of the *eff* function in the form of a abstract syntax tree traverser.

This information suffices the framework to verify a new effect domain.

Exceptions. The implementation of an effect system for exceptions was as straightforward as expected. Empirical validation shows that the annotation overhead is greatly reduced compared to the non-polymorphic `throws` clauses found in Java. The work on anchored exceptions [22] gives more detailed insights to research in this area.

There are however a few cases where a static effect analysis cannot compute the possibly thrown exceptions of an expression correctly. Even though these limitations are unrelated to the polymorphic effect checking framework presented in this paper, we still mention them for completeness.

To make an example, assume that the method `head` of a list class throws an exception when the list is empty. If we analyze the method

```
def headOrZero(l: List[Int]) = if (l.isEmpty) 0 else l.head
```

we clearly see that the invocation `l.head` will not throw an exception. However, the effect system does not take conditional control flow into account and will therefore conclude that the expression might throw an exception.

In Java, this specific problem is bypassed by having unchecked exceptions, a class of exception types which are not tracked by the effect system. The `NoSuchElementException` in Java is such an unchecked exception, and therefore the example expression would be treated as pure.

Another solution we are considering is to allow programmers to override the inferred effects by introducing syntax for effect casting. We think that this can prevent programmers from disabling effect checking altogether just because in some situations, the analysis computes an imprecise result.

5 Related Work

Polymorphic effect systems are introduced in [13] as part of the FX programming language [7]. The overhead of having explicit effect and region parameters is overcome by later work on type and effect inference [21]. However, type reconstructions requires the whole program to be known and does not allow modular reasoning about effects. Our approach allows modular effect checking using latent effect annotations and effect-polymorphic method types. It also enables more immediate feedback while developing an application, for instance in an IDE.

The work on anchored exceptions [22] extends the `throws` annotations in Java's checked exceptions to express effect-polymorphism. Instead of listing the concrete exception types that might be thrown, an annotation can take the form `throws like a.m()` where `a` is a parameter of the method. Their system is tied to checked exceptions, however we believe that the main ideas can be equally applied in a generic system. In order to do so, our type system could be extended with dependent types, allowing effect annotations to refer to parameter names.

Marino et al. [14] describe a generic type-and-effect system that can be instantiated to different effect domains. Our effect framework can be seen as an extension of their work with effect polymorphism. There are a few differences however; most importantly they support a tagging system for run-time values. Reconstructing which tags can flow into a function argument uses a whole-program analysis; the authors refer to type qualifier inference [4]. Our system is designed to work modularly on the basis of annotations.

There exist a number of other approaches than type-and-effect systems to delimit the scope of side effects. The most common alternative is monadic encapsulation of effects, which has been shown to be equivalent to effect systems by Wadler et al. in [23]. One aspect of type-and-effect systems shown in Section 3.1 is that combining multiple effect domains is straightforward, while composing monads on the other hand is difficult [11]. Applicative functors [6] are one promising upcoming alternative to monads which facilitate composition.

6 Conclusions and Future Work

We designed an extensible framework for polymorphic effect checking where multiple effect domains can be integrated modularly. Annotating functions as effect-polymorphic is lightweight in syntax and independent of specific effect domains. We implemented the framework for the Scala programming language in the form of a compiler plugin and successfully applied polymorphic effect checking to real-world examples such as the Scala collections library.

We are studying two directions to make the presented type-and-effect system more expressive. First, we investigate how annotations for effect-polymorphism can be generalized using dependent types. This extension allows us to express effect masking behavior in function types, and to denote the behavior of methods more precisely in the object-oriented case by stating which members of the arguments contribute to effect-polymorphism.

As a second extension, we are studying a generalization to support flow-sensitive effects, so that more effect systems can be expressed as a plugin to our framework. One example of a flow-sensitive effect system is the purity analysis presented in [18].

When integrating a type-and-effect system into an existing programming language, there are also practical issues that need consideration. One question is how to handle the interaction with legacy code that does not provide effect annotations. A possible solution is to consider whole-program effect analysis techniques to reconstruct effect annotations for existing libraries.

References

1. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 63–74. ACM, New York (2008)
2. Bocchino Jr., R.L., Adve, V.S.: Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 306–332. Springer, Heidelberg (2011)
3. Bracha, G.: Pluggable type systems. In: OOPSLA 2004 Workshop on Revival of Dynamic Languages (2004)
4. Foster, J.S., Johnson, R., Kodumal, J., Aiken, A.: Flow-insensitive type qualifiers. ACM Trans. Program. Lang. Syst. 28, 1035–1087 (2006)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
6. Gibbons, J., Oliveira, B.C.D.S.: The essence of the iterator pattern. In: McBride, C., Uustalu, T. (eds.) Mathematically-Structured Functional Programming (2006)
7. Gifford, D.K., Jouvelot, P., Sheldon, M.A., O'Toole, J.W.: Report on the FX programming language. Technical report (1992)
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java(TM) Language Specification, 3rd edn. Addison-Wesley Professional (2005)
9. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3), 202–220 (2009)
10. Hejlsberg, A.: The trouble with checked exceptions (2003), http://www.artima.com/intv/handcuffs.html
11. King, D., Wadler, P.: Combining monads. In: Mathematical Structures in Computer Science, pp. 61–78 (1992)
12. Lea, D.: A Java fork/join framework. In: Java Grande, pp. 36–43 (2000)
13. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 47–57. ACM, New York (1988)
14. Marino, D., Millstein, T.: A generic type-and-effect system. In: Proceedings of the 4th International Workshop on Types in Language Design and Implementation, TLDI 2009, pp. 39–50. ACM, New York (2009)
15. Mikhailova, A., Romanovsky, A.: Supporting evolution of interface exceptions, pp. 94–110. Springer-Verlag New York, Inc., New York (2001)
16. Odersky, M.: The Scala language specification (2011), http://www.scala-lang.org/docu/files/ScalaReference.pdf

17. Odersky, M., Moors, A.: Fighting bit rot with types (experience report: Scala collections). In: Kannan, R., Narayan Kumar, K. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009). Leibniz International Proceedings in Informatics (LIPIcs), vol. 4, pp. 427–451. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl (2009)
18. Pearce, D.J.: JPure: A Modular Purity System for Java. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 104–123. Springer, Heidelberg (2011), doi:10.1007/978-3-642-19861-8_7
19. Rytz, L., Odersky, M.: Lightweight polymorphic effects - proofs. Technical report, EPFL (2012)
20. Syme, D., Petricek, T., Lomov, D.: The F# Asynchronous Programming Model. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 175–189. Springer, Heidelberg (2011)
21. Talpin, J.-P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming 2(3), 245–271 (1992)
22. van Dooren, M., Steegmans, E.: Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 455–471. ACM, New York (2005)
23. Wadler, P., Thiemann, P.: The marriage of effects and monads. ACM Trans. Comput. Logic 4, 1–32 (2003)

Cloud Types for Eventual Consistency

Sebastian Burckhardt[1], Manuel Fähndrich[1], Daan Leijen[1], and Benjamin P. Wood[2]

[1] Microsoft Research
[2] University of Washington

Abstract. Mobile devices commonly access shared data stored on a server. To ensure responsiveness, many applications maintain local replicas of the shared data that remain instantly accessible even if the server is slow or temporarily unavailable. Despite its apparent simplicity and commonality, this scenario can be surprisingly challenging. In particular, a correct and reliable implementation of the communication protocol and the conflict resolution to achieve eventual consistency is daunting even for experts.

To make eventual consistency more programmable, we propose the use of specialized cloud data types. These cloud types provide eventually consistent storage at the programming language level, and thus abstract the numerous implementation details (servers, networks, caches, protocols). We demonstrate (1) how cloud types enable simple programs to use eventually consistent storage without introducing undue complexity, and (2) how to provide cloud types using a system and protocol comprised of multiple servers and clients.

1 Introduction

As technology progresses, new applications emerge. Of growing popularity are downloadable applications, so-called *apps*, that offer specialized functionality on a mobile device such as a phone or a tablet. Often, these apps include social aspects where users share information online. The capability of sharing data between devices is typically achieved by developing custom webservices. Increasingly, such services are deployed in the *cloud*, hosted environments that offer virtualized storage and computing resources.

Some apps require synchronization of data among multiple devices by the same users. For example, users may want to share settings, calendars, contact lists, or personal music databases. Other apps empower data sharing and communication among multiple users. For example, a simple grocery list can help family members keep track of items to be purchased on the next trip to the store. Moreover, many apps can benefit from including social features that let users share information, comments, reviews, achievements, high scores, and so on.

A pervasive requirement for apps is that they remain responsive at all times. Unfortunately, server connections are notoriously prone to slowness or temporary

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 283–307, 2012.

outages. Another important requirement is that apps consume little battery power, and do not transfer much data. Thus, well-engineered apps must avoid reliance on excessive server communication.

A common technique is to maintain a replica of the data on each device. Since this replica is always available for queries and updates, apps remain responsive even if the device is disconnected. When reconnected, updates can be propagated to all other replicas in such a way that the resulting data is *eventually consistent* [12,11]. Not only does this ensure responsiveness, but users can limit connections to times where power or bandwidth are ample (such as while the device is charging at home and connected to a home network).

Clearly, sharing data via eventually consistent local replicas is an attractive solution for many applications; unfortunately, it can be daunting to implement. Typical challenges include:

- *Representation.* Because app programming is at the intersection of historically separate communities, programmers often end up writing and maintaining inordinate amounts of code to translate between different data representations (SQL, HTTP, JSON, XML, object heaps). Moreover, app programmers are forced to write custom web services, and may have to deal with subtle programming platform differences between clients and servers.
- *Consistency.* Since multiple devices can update their local replicas at the same time while disconnected, clients can detect conflicts only after the fact, when sending changes to the server. When such transactions fail, one must write code explicitly to resolve the conflict. For example, if several users update the same entry in a grocery list, we must be careful not to lose updates.
- *Change sets.* Support for disconnected operation typically means that an app must store not just a local replica, but also log a delta of all the updates that are performed locally. When the device is reconnected, these are the updates that are now sent to the server replica. Reliably resolving conflicting operations inside a large change set can be a difficult problem.

Given these hurdles and challenges it is not surprising that many apps do not implement our requirements fully: for example, updates can often only be performed while connected, and the app blocks while the transaction takes place. Other apps allow non-blocking updates but do not guarantee eventual consistency.

To make it easier for app developers to share data in the cloud, we propose the use of *cloud types* at the programming language level. Cloud types provide an abstraction layer that frees app developers of the recurring engineering challenges (web service implementation, communication protocol, local storage) and allows them to focus on the essentials: declaring the data strucures and writing client code that accesses them. The two main ingredients of our solution are:

1. *Cloud Types.* Programmers declare the data they wish to share using special cloud types. This data is automatically shared between all devices, and is automatically persisted both on local storage as well as in cloud storage. Our cloud types include both simple types (cloud integers, cloud strings)

and structured types (cloud arrays, cloud entitities). Because cloud types are carefully chosen to behave predictably under concurrent modfication, conflict resolution is automatic and the developer need not write special code to handle merging.

2. *Revision Consistency.* Our system uses *revision diagrams* to guarantee eventual consistency, as proposed in [1]. Conceptually, the cloud stores the main revision, while devices maintain local revisions that are periodically synchronized. Revision diagrams are reminiscent of source control systems and provide an excellent intuition for reasoning about multiple versions and eventual consistency.

Our approach integrates all aspects of the data model (declarations, queries, and updates) directly into the programming language. Thus, there is only one program and only one data format. Code can read and modify the data directly, without buffering or copying, and without blocking. Note that we do not propose a new language as such. Many existing languages could be extended seamlessly with support for our data model, or use libraries to expose the functionality. Currently, we have a partial implementation of our model in the TouchDevelop [17] language and development environment.

Although cloud servers are used to maintain consistency, the app developer does not write any code that executes on the server. The data declarations completely determine the functionality of the server. For the purpose of this paper, we leave out session and authorization management as they can be dealt with separately.

Overall, we make the following contributions:

- We present a data model that directly integrates support for eventually consistent data into the programming language. We demonstrate how this model allows us to write simple programs for common scenarios by walking through several examples (Section 2).
- We show how the data schema can be composed from basic cloud types (Section 2.4), and how advanced cloud types (such as observed-remove sets [13]) can be built up from simpler ones (Section 5).
- We provide a comprehensive formal syntax and semantics. It connects a small, but sufficiently expressive programming language (Section 3) with a detailed operational model of a distributed system containing a server-pool and multiple client devices (Section 4). These models are connected by a fork-join automaton (an abstract data type supporting eventual consistency) derived automatically from the schema (Section 5). Together, these models extend and concretize earlier work on eventually consistent transactions [1].

2 Overview

In this section we introduce our model in more detail, by gradually introducing the basic cloud types (cloud integers, cloud strings, cloud arrays, and cloud entitites) using examples. Along the way, we explain the execution model (revision diagrams) and the synchronization primitives (*yield*, *flush*). We start with a

simple grocery list, followed by a customer database, and conclude with a seat reservation example.

For the purposes of this paper, all examples are in pseudocode using a typed javascript-like language. We believe that the essential features of our system can be incorporated in most real-world static or dynamic languages in a seamless way. We currently have a partial implementation directly in the TouchDevelop language and as a library in C#.

2.1 Cloud Integers and Cloud Arrays

A simple but quite common scenario is the ever popular grocery list application found on all mobile devices. We show the code for this example in Fig. 1, and now proceed to explain it in detail.

```
// declaration of cloud data

global totalItems : CInt;

array Grocery[ name : String ]
{
  toBuy : CInt;
}

// operations representing user actions

function ToBuy( name : String, count : Int )
{
  totalItems.add(count);
  Grocery[name].toBuy.add(count);
}

function Bought( name : String, count : Int )
{
  totalItems.add(− count);
  Grocery[name].toBuy.add(− count);
}

function Display()
{
  foreach g in entries Grocery.toBuy
  {
    Print(g.toBuy.get() + " " + g.name);
  }
  Print(totalItems.get() + " total");
}

// main event loop

function main()
{
  bool done = false;
  while (not done)
  {
    //allow send/receive of updates
    yield();

    match (NextUserCommand()) with
    {
      buy s n:
        ToBuy(s, n);
      bought s n:
        Bought(s, n);
      display:
        Display();
      quit:
        done = true;
    }
  }
}
```

Fig. 1. Pseudocode for the grocery list example

First, consider the cloud data declarations (Fig. 1, left column, top). We use the term "cloud data" to emphasize that the data declared in this section is automatically replicated across all devices. One could say we are declaring *global variables, literally.* For this example, we chose to store both (1) a count of the total groceries to buy, and (2) a count for each individual grocery item. Although not really important for this example, storing the total count is helpful to illustrate the consistency model later on.

- To represent the total count, we declare a variable called TotalCount of type **CInt**. This type is a primitive datatype for storing and manipulating cloud integers. It differs from ordinary integer variables in that it offers *higher level operations* that have better conflict resolution semantics than using get and set operations alone. In particular, it offers an add operation to express a *relative* change, reminiscent of atomic or interlocked instructions in shared-memory programming.
- To represent the quantity of each item, we use a *cloud array* called **Grocery**. The array is indexed by the name of the grocery item, and each entry stores the quantity toBuy. This quantity is again of type **CInt**.

Cloud arrays are a bit different from standard arrays as the index type can be infinite (as in this case, strings). Cloud array entries can have multiple fields, although there is only one in this example (toBuy). Moreover, all array entries are always defined, and we guarantee that all fields are initialized with the default value (which is 0 for **CInt**). Next, consider the actions on the data (Fig. 1)

- (ToBuy) When adding count items of name name, we adjust both the total totalItems as well as the specific item count stored in the array, using the primitive add which is supported by the cloud integer type. To access the array entry, we use the name of the cloud array (**Grocery**) and an index [name] and field (toBuy).
- (Bought) When removing items from the list, we proceed the same way, but subtract the quantity rather than adding it.
- (Display) To display the list, we need to iterate over the array. This requires a bit of extra thought since we cannot just iterate over all (infinitely many) strings. Rather, we iterate over *entries* **Grocery**.toBuy, which returns only array entries for which the field toBuy is not the default value (0 for **CInt**). Thus, we print the name and the count of all items for which the count is not zero.

Finally, consider the pseudocode for the main program (Fig. 1, right column). Since the grocery list is an interactive program, it executes some form of loop to handle user commands.[1] The important part is the *yield* statement. Essentially, the *yield* statement gives the runtime system the permission to both (1) propagate changes made locally to the replica to other devices, and (2) apply changes made by other devices to the local replica. *yield* is nonblocking and guaranteed

[1] In a realistic event-based application framework the API is likely to be different, but we believe our simple loop is sufficient to convey the idea.

to execute very quickly. *yield* does not force synchronization: it is perfectly acceptable for *yield* to do nothing at all (which is in fact all it can do in situations where the device is not connected).

Another way to describe the effect of *yield* is that the *absence* of a yield guarantees isolation and atomicity; *yield* statements thus partition the execution into a form of transaction (called eventually consistent transactions in [1]). Effectively, this implies that everything is always executing inside a transaction. The resulting atomicity is important for maintaining invariants; in this example, it guarantees that the total count is always equal to the sum of all the individual counts, since all changes made to **Grocery** and totalCount are always applied atomically.

2.2 Revision Diagrams and Cloud Types

Now that we have sketched some basic language features, it is time to explain the execution model in more detail. Our semantics are based on concurrent revisions [2], and rely on the following main concepts:

- *Revision diagrams* are reminiscent of source control systems, and show the order in which revisions are forked and joined. Conceptually, each revision keeps a log of all the updates that were performed in it. When a revision is joined into another revision, it replays all logged updates into that revision.
- *Cloud types* are abstract data types that offer a precisely defined collection of update and query operations. Moreover, cloud types can provide optimized fork and join implementations and space-bounded representations of logs.

For example, consider a cloud variable x of type **CInt** and the revision diagram examples in Fig 2. Note that join is not symmetric; join orders updates of the joined revision *after* the updates of the joining revision, and update operations are not always commutative (for example, two add operations do commute, but set operations do not commute).

2.3 Execution Model and Eventual Consistency

We can now employ revision diagrams to build an eventually consistent distributed system. The idea is to keep the main revision on the server, and to keep some revision always available on each device, whether connected or not. We can

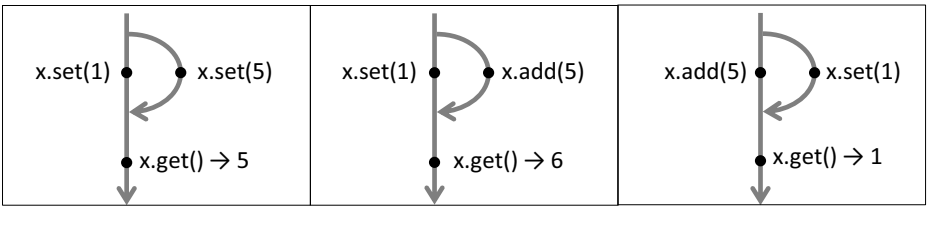
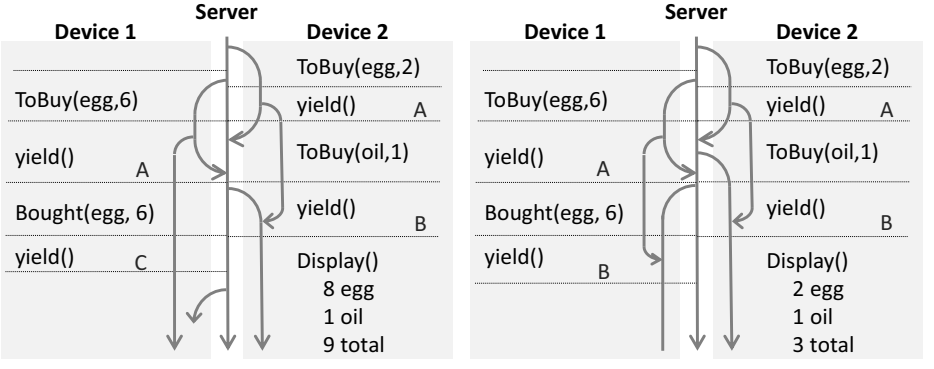


Fig. 2. Conflict resolution for **CInt**. Updates of the revision are replayed at the join point.

send revisions from the server to clients, and vice versa, and perform forks and joins on either one.

Program execution is nondeterministic if multiple devices are involved.[2] Both of the following revision diagrams represent possible executions of the grocery list example. Because of timing differences, the Display() on device 2 may either see the first update by device 1 (left) or not see it (right).



Forking and joining of revisions on the server is straightforward. The implementation of *yield* on the device is a bit more clever, since it is guaranteed to always execute quickly and never block (regardless of message speed or lost messages). We achieve this by distinguishing 3 cases (labeled A,B,C in the executions above): (A) If we are not currently expecting a server response, we send the current revision to the server, after forking a new revision for continued local use. (B) If a revision from the server has arrived, we merge the current local revision into it. (C) If we are expecting a revision from the server but it is not present yet, we do nothing.

As long as clients repeatedly call *yield*, and as long as messages are eventually delivered (using retransmission if necessary), eventual consistency is achieved. We present formal operational semantics for *yield* in Section 4.

Since revision diagrams are quite general, a wide variety of implementation choices beyond the one sketched above can be employed (such as servers organized as trees, or clients bundled with servers connected peer-to-peer). We discuss one specific multi-server implementation model (a server pool) in Section 4.

2.4 Entities

Our model is versatile enough to store complex relational data. In Fig. 3, we consider a mobile application that maintains a database of customers and orders (as may be used by a small business in an emerging market).

[2] Determinism makes no sense for eventually consistent systems, since such systems are expected to adapt opportunistically to unpredictable message latency and loss.

```
entity Customer
{
  name : CString
}
array Product[ id : string ]
{
  name : CString
  price : CInt
}
entity Order(customer : Customer)
{
  time : CTime
  totalprice : CInt
}
array CartItem [
  customer : Customer,
  product : Product ]
{
  quantity : CInt;
}
array OrderItem [
  order : Order,
  product : Product ]
{
  quantity : CInt
  price : CInt
}
function AddToCart(c: Customer,
      p: Product, q: int)
{
  CartItem[c,p].quantity.add(q);
}
function DeleteCustomer(c: Customer)
{ delete c; }
```

```
function SubmitOrder(customer : Customer)
{
  // create fresh order
  var order = new Order(customer);
  order.time.set(now());
  // move items from cart
  foreach cartitem in entries CartItem.quantity
    where cartitem.customer == customer
  {
     var oitem = OrderItem[order, cartitem.product];
     oitem.quantity = cartitem.quantity;
     oitem.price = cartitem.quantity
                   * cartitem.product.price;
     cartitem.quantity.add(-oitem.quantity);
     order.totalprice.add(oitem.price);
  }
}

function ShowOrders(customer : Customer)
{
  foreach order in all Order
    where order.Customer == customer
    orderby order.time
  {
    Print("Order of " + order.time);
    foreach(i in all OrderItem)
      where i.order == order
    {
       Print(i.quantity + " " + i.product.name +
             " for " + i.price);
    }
  }
}
```

Fig. 3. Pseudocode for the customer database example

Since arrays do not support dynamic creation or deletion of entries, we introduce an alternative form of data structures, called entities.[3] We model customers and orders as entities rather than array entries, which has two advantages: (1) we can create them without first determining an index by which to identify them uniquely, and (2) we can delete them explicitly, which removes them (as well as all associated data) from the database.

[3] Note that both our arrays and our entities are special cases of the general notion of entities as used in Chen's entity-relationalship model [3]. The distinction is that our array entries have visible primary keys (the indexes), and can not be created or deleted, while our entities have hidden, automatically managed primary keys, and are explicitly created and deleted by the user.

Product is an array of products, indexed by a unique id, and **CartItem** is an array of cart items, indexed by customers and products, storing the quantity. The entity **Order** takes a customer as a construction argument (construction arguments are like immutable fields, but also play an additional role explained below), and the array **OrderItem** stores the quantity of each product in each order.

The function AddToCart adds items to a customer's cart, just as we added items to the grocery list in the previous example. The function SubmitOrder creates a new order entity for the customer, then iterates through the cart items of this customer, and adds them to the order, totaling the prices. Note that since there is no *yield* in this function, we need not worry about the order entity becoming visible to other devices before all of its information is computed. The function ShowOrders prints all orders by a customer, sorted by date. It uses the query *all* **Order** *where* order.**Customer** == order which returns all order entities belonging to this customer.

The function DeleteCustomer is simple, but has some interesting effects. Not suprisingly, it deletes the customer entity. But beyond that, it also clears all entries in all arrays that have the deleted customer as an index, and it even deletes all orders that have the deleted customer as a construction argument.[4]

2.5 Stronger Consistency

Eventual consistency is not always sufficient. Some problems, such as reserving a seat on an airplane, or withdrawing money from a bank account, involve a limited resource and require true arbitration. In such cases, we *must* establish a server connection and wait for a response. In this section, we show how to reintroduce strong synchronization.

Consider an application making seat reservations, which may attempt something like the following:

```
array Seat [
  row : int,
  letter : string ]
{
  assignedTo : CString;
}
```

```
function NaiveReserve(seat: Seat, customer : string)
{
  if (seat.assignedTo.get() == "")
    seat.assignedTo.set(customer);
  else
    print("reservation failed");
}
```

Unfortunately, this does not work as desired: a seat may appear empty in the local revision, but already be filled on the server. In this case, the NaiveReserve function would appear to succeed, but in fact may overwrite another reservation once the update reaches the server. We fix this problem by introducing a primitive operation setIfEmpty for the cloud type **CString**. This operation sets the string only if it is currently empty, and this condition is reevaluated when the update operation is applied on the server. Thus, existing reservations are never overwritten.

[4] Entities whose existence depends on other entities are sometimes called 'weak entities' in the literature. In our system, those 'weak entities' correspond to (1) entities that have other entities appearing in their construction arguments, and (2) array entries that have entities appearing as an index.

However, *yield* is still not sufficient to force mutual exclusion, since we can not tell when the update has reached the server. Thus we support an additional synchronization primitive called *flush*. Upon *flush*, execution blocks until (1) all local updates have been applied to the main revision, and (2) the result has become visible to the local revision. Now we can implement the body of the reservation function as follows:

```
seat.assignedTo.setIfEmpty(customer);
flush;
if (seat.assignedTo.get() ≠ customer) print("reservation failed");
```

Since *flush* could block indefinitely if the device is not connected, our implementation supports the specification of a timeout.

This example is interesting since it shows that our model is at least as expressive as shared-memory programming with locks (locks can be implemented analogously). However, it does not represent the type of application for which our model is most suited. On the contrary, for applications that frequently require strong synchronization, the benefits of our model are marginal, and traditional OLTP (online transaction processing) is likely more appropriate.

3 Syntax, Types, and Local Semantics

Figure 4 describes the syntax of types, schemas, and expressions. We distinguish three kinds of types. The index type ι is the type of values that can be used as indices into an array or entity, and consists of simple read-only values like **Int**, **String**, and array and entity identifiers (A and E). The cloud type ω is used for mutable cloud values that are persisted. We prefix such types with the letter **C** to distinguish them from regular value types. Examples of cloud types are **CInt** and **CString**. In Section 5 we give precise semantics to these cloud values using *fork-join automata*. The type **CSet**$\langle\iota\rangle$ is the type of *observed-remove* sets as described by Shapiro [13]. Finally, we have expression types τ which includes index types ι, functions, products, and regular sets. We denote the trivial product ($n = 0$) by **Unit**.

A schema $\mathcal{S}$ consists of a sequence of declarations. A declaration is either an array A, an entity E, or a property p. Properties map an index ι to a mutable cloud type ω. In our examples, we used the following syntactic sugar to define properties as part of an array or entity declaration:

$$\textit{entity}\ E(k_1 : \iota_1, ..., k_m : \iota_m)\ \{\, p_1 : \omega_1, ..., p_n : \omega_n \}$$
$$\equiv\ \textit{entity}\ E(k_1 : \iota_1, ..., k_m : \iota_m);\ \textit{property}\ p_1 : E \to \omega_1;\ ...;\ \textit{property}\ p_n : E \to \omega_n$$

$$\textit{array}\ A[k_1 : \iota_1, ..., k_m : \iota_m]\ \{\, p_1 : \omega_1, ..., p_n : \omega_n \}$$
$$\equiv\ \textit{array}\ A[k_1 : \iota_1, ..., k_m : \iota_m];\ \textit{property}\ p_1 : A \to \omega_1;\ ...;\ \textit{property}\ p_n : A \to \omega_n$$

Also, global persisted values (as used for example in the grocery list in Figure 1) are syntactic sugar for cloud arrays without any keys and a single value property:

$$\textit{global}\ x : \omega\ \equiv\ \textit{array}\ x[\,]\ \{\, \mathit{value} : \omega \}$$

entity names	$Ent \ni$	E	::= ...
array names	$Arr \ni$	A	::= ...
index types		ι	::= **Int** | **String** | E | A
cloud types		ω	::= **CInt** | **CString** | **CSet**$\langle\iota\rangle$ | ...
expression types		τ	::= ι | **Set**$\langle\tau\rangle$ | $\tau \to \tau$ | $(\tau_1, ..., \tau_n)$
key names		k	::= ...
property names		p	::= ...
declarations		$decl$	::= *entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$ | *array* $A[k_1 : \iota_1, ..., k_n : \iota_n]$ | *property* $p \ : \ \iota \to \omega$
schema		$\mathcal{S}$	::= $decl_1; ...; decl_n$
unique id's	$Uid \ni$	uid	::= ... (abstract)
constants	$Con \ni$	c	::= ... (integer and string literals)
updates		op_u	::= ... (predefined)
queries		op_q	::= ... (predefined)
operations		op	::= op_u | op_q
values	$Val \ni$	v	::= $A[v_1, ..., v_n]$ | $E[uid, v_1, ..., v_n]$ | c | x | $(v_1, ..., v_n)$ | $\lambda(x : \tau).\, e$
expressions		e	::= *new* $E(e_1, ..., e_n)$ | *delete* e | $A[e_1, ..., e_n]$ | $e.p.op(e_1, .., e_n)$ | $e.k$ | *all* E | *entries* p | *yield* | *flush* | v | $e_1\, e_2$ | $e_1; e_2$ | $(e_1, ..., e_n)$
program		$program$	::= $\mathcal{S}; e$

Fig. 4. Syntax of types, schemas, and expressions. A subscript n without an explicit bound is assumed $n \geqslant 0$.

where all operations on x are replaced with operations on the array value:

$$x.op(e_1, ..., e_n) \ \equiv \ x[\,].value.op(e_1, ..., e_n)$$

The syntax of expressions is separated into values v and expressions e to facilitate the description of the evaluation semantics. Values can be regular values such

as literals c, variables x, products of values, or lambda expressions. Moreover, we have array and entity values which encode a particular entry of an array, as $A[v_1, ..., v_n]$, or a particular entity as $E[uid, v_1, ..., v_n]$. The entity value is not an expression that users can write down themselves and only occurs in the evaluation semantics as the result of a *new* expression (which also supplies the unique id uid for the entity value).

Expressions consist of both cloud specific expressions, and of regular expressions like applications $e_1\, e_2$, sequence $e_1; e_2$, products and lambda expressions. The keywords *new* and *delete* respectively create and delete entities. The expression $A[e_1, ..., e_n]$ is used to index into an array. The operation expression $e.p.op(e_1, ..., e_n)$ invokes an update or query operation op on a property p indexed by e. The creation keys of an entity, or the indices of an array expression can be queried using the $e.k$ expression.

The *all* and *entries* keywords return all elements of an entity or all non-initial entries of a property respectively. These primitive expressions allow us to construct general queries. Finally, the *yield* and *flush* operations are used for synchronization with the cloud.

Figure 5 defines a type system for our expression language. A derivation $\mathcal{S}, \Gamma \vdash e : \tau$ states that for a certain (well-formed) schema $\mathcal{S}$ and type environment Γ, the expression e is well-typed with a type τ. The initial Γ is written as Γ_0 and contains the type of primitive functions (i.e. $add : (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}$), together with the types of primitive cloud type operations (i.e. $\mathsf{CInt}.add : (\mathsf{Int}) \to \mathsf{Unit}$).

Most rules are standard and self-explanatory. There are some important details though. In particular, in the type rule for operation expressions, we can see that the type ω of the mutable cloud value never 'escapes': values with a cloud type ω are not first-class and expressions always have a type τ (which does not include ω). This is by construction since an operation expression $e.p.op(e_1, ..., e_n)$ always occurs as a bundle and the cloud value never occurs in isolation.

3.1 Client Execution

Figure 6 and 7 give the evaluation semantics for local client execution. Figure 6 defines the evaluation order within an expression. An execution context $\mathcal{E}$ is an expression "with a hole $\square$", and we use the notation $\mathcal{E}[\![e]\!]$ to denote the expression obtained from $\mathcal{E}$ by replacing the hole with e. Essentially, the execution context acts as an abstraction of a program counter and specifies where the next evaluation step can take place.

Figure 7 defines the operational semantics in the form of transition rules $e; \sigma \to e'; \sigma'$ where an expression e with a local state σ is evaluated to a new expression e' and updated local state σ'. The client state σ is the state of the schema fork-join automaton $\Sigma^{\mathcal{S}}$ described in Section 5.

The first three rules, *new*, *delete*, and operation expressions just update the local state by invoking the corresponding updates on the fork-join automaton. The following three query rules just return the result of executing the corresponding query on the fork-join automaton. The fresh uid for the `create` call is produced locally. We assume each client can generate such globally unique ids.

$$\frac{\mathit{entity}\ E(k_1 : \iota_1, ..., k_n : \iota_n) \in \mathcal{S} \quad \mathcal{S}, \Gamma \vdash e_i : \iota_i}{\mathcal{S}, \Gamma \vdash \mathit{new}\ E(e_1, ..., e_n) : E} \qquad \frac{\mathcal{S}, \Gamma \vdash e : E}{\mathcal{S}, \Gamma \vdash \mathit{delete}\ e : \mathbf{Unit}}$$

$$\frac{\mathit{array}\ A[k_1 : \iota_1, ..., k_n : \iota_n] \in \mathcal{S} \quad \mathcal{S}, \Gamma \vdash e_i : \iota_i}{\mathcal{S}, \Gamma \vdash A[e_1, ..., e_n] : A} \qquad \frac{\mathit{entity}\ E(...) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash E[uid, v_1, ..., v_n] : E}$$

$$\frac{\mathcal{S}, \Gamma \vdash e : \iota \quad \mathit{property}\ p : \iota \to \omega \in \mathcal{S} \quad \omega.op : (\tau_1, ..., \tau_n) \to \tau \in \Gamma \quad \mathcal{S}, \Gamma \vdash e_i : \tau_i}{\mathcal{S}, \Gamma \vdash e.p.op(e_1, ..., e_n) : \tau}$$

$$\frac{\mathit{entity}\ E(...) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash \mathit{all}\ E : \mathbf{Set}\langle E \rangle} \qquad \frac{\mathcal{S}, \Gamma \vdash e : E \quad \mathit{entity}\ E(..., k : \iota, ...) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash e.k : \iota}$$

$$\frac{\mathit{property}\ p : \iota \to \omega \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash \mathit{entries}\ p : \mathbf{Set}\langle \iota \rangle} \qquad \frac{\mathcal{S}, \Gamma \vdash e : A \quad \mathit{array}\ A[..., k : \iota, ...] \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash e.k : \iota}$$

$$\frac{x : \tau \in \Gamma}{\mathcal{S}, \Gamma \vdash x : \tau} \qquad \frac{\mathcal{S}, (\Gamma, x : \tau_1) \vdash e : \tau_2}{\mathcal{S}, \Gamma \vdash \lambda(x : \tau_1).\, e\ : \tau_1 \to \tau_2}$$

$$\frac{\mathcal{S}, \Gamma \vdash e_1 : \tau_2 \to \tau \quad \mathcal{S}, \Gamma \vdash e_2 : \tau_2}{\mathcal{S}, \Gamma \vdash e_1\, e_2 : \tau} \qquad \frac{\mathcal{S}, \Gamma \vdash e_i : \tau_i}{\mathcal{S}, \Gamma \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)}$$

$$\frac{\mathcal{S}, \Gamma \vdash e_1 : \tau_1 \quad \mathcal{S}, \Gamma \vdash e_2 : \tau_2}{\mathcal{S}, \Gamma \vdash e_1; e_2 : \tau_2} \qquad \frac{}{\mathcal{S}, \Gamma \vdash \mathit{yield} : \mathbf{Unit}} \qquad \frac{}{\mathcal{S}, \Gamma \vdash \mathit{flush} : \mathbf{Unit}}$$

Fig. 5. Types of expressions

$$\begin{aligned}
\mathcal{E} ::= {} & \Box \\
| \ & \mathit{new}\ E(v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n) \\
| \ & \mathit{delete}\ \mathcal{E} \\
| \ & A[v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n] \\
| \ & \mathcal{E}.p.op(e_1, ..., e_n) \\
| \ & v.p.op(v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n) \\
| \ & \mathcal{E}.k \\
| \ & \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathcal{E}; e \\
| \ & (v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n)
\end{aligned}$$

Fig. 6. Evaluation contexts

The final four rules are standard evaluation rules on the expressions and do not use the local state at all. Note that we chose to keep the creation keys of arrays and entities around explicitly in the value representation which makes the key selection a completely local operation. However, realistic implementations can use just the *uid* to represent entities and store the creation values in the local state (and similarly for arrays).

The operations *yield*, *flush*, and *barrier* cannot be described as local operations and are handled by the semantic rules defined over the clients and servers as shown in the next section.

$$
\begin{array}{lcll}
\mathcal{E}[\![\textsf{new}\, E(v_1,...,v_n)]\!];\sigma & \to & \mathcal{E}[\![E[uid, v_1,...,v_n]]\!];\sigma.\mathsf{create}_E(E[uid,v_1,...,v_n]) & \text{(fresh } uid\text{)} \\
\mathcal{E}[\![\textsf{delete}\, E[uid,...]]\!];\sigma & \to & \mathcal{E}[\![()]\!];\sigma.\mathsf{delete}_E(uid) & \\
\mathcal{E}[\![v.p.op_u(v_1,...,v_n)]\!];\sigma & \to & \mathcal{E}[\![()]\!];\sigma.\mathsf{update}_p(v, op_u(v_1,...,v_n)) & \\
 & & & \\
\mathcal{E}[\![v.p.op_q(v_1,...,v_n)]\!];\sigma & \to & \mathcal{E}[\![\sigma.\mathsf{query}_p(v, op_q(v_1,...,v_n))]\!];\sigma & \\
\mathcal{E}[\![\textsf{all}\ E]\!];\sigma & \to & \mathcal{E}[\![\sigma.\mathsf{all}_E]\!];\sigma & \\
\mathcal{E}[\![\textsf{entries}\ p]\!];\sigma & \to & \mathcal{E}[\![\sigma.\mathsf{entries}_p]\!];\sigma & \\
 & & & \\
\mathcal{E}[\![A[v_1,...,v_n].k_i]\!];\sigma & \to & \mathcal{E}[\![v_i]\!];\sigma & \\
\mathcal{E}[\![E[uid,v_1,...,v_n].k_i]\!];\sigma & \to & \mathcal{E}[\![v_i]\!];\sigma & \\
\mathcal{E}[\![(\lambda(x:\tau).e)\, v]\!];\sigma & \to & \mathcal{E}[\![e[v/x]]\!];\sigma & \\
\mathcal{E}[\![v;e]\!];\sigma & \to & \mathcal{E}[\![e]\!];\sigma &
\end{array}
$$

Fig. 7. Expression semantics

4 System Model and Distribution

In the previous section, we have established the local execution semantics of expressions. In this section, we present an operational whole-system model including multiple clients and an elastic server pool. We follow the general blueprint for modeling eventually consistent systems presented in [1], where we prove that to achieve eventual consistency, it is sufficient to enforce that all executions produce proper revision diagrams, and that we use proper fork and join functions to manage the state of replicas.

Fig. 8(a) shows a brief example of an execution with three servers in the pool, and two clients. Clients that perform *yield* or *flush* initiate transitions of two kinds, push and pull. These transitions communicate with an eligible server in the pool. Not all servers are eligible, as we will explain shortly. Servers behave similarly to clients, initiating push and pull transitions with other eligible servers.

When synchronizing, clients and servers need to ensure that proper revision diagrams result. In particular, they must observe the join rule [1]: joiners must be downstream from the fork that forked the joinee (see Fig. 8b for examples). To ensure this condition, we assign round numbers to servers and clients, and use round maps (a form of vector clocks) to determine eligibility (by determining which forks are in the visible history). We show round numbers in Fig. 8(a). All clients and servers start with round 0, except the main revision that starts (and forever remains) in round 1.

We now proceed to give formal definitions of the ideas outlined above. We begin by introducing some notation to prepare for the operational rules in Fig. 9 and Fig. 10. We define a system configuration $\mathcal{C}$ to be a partial function from identifiers (representing servers or clients) to a server or client state, respectively. For a client identifier c, the client state $\mathcal{C}(c)$ is a tuple (r, e, σ) consisting of a round number r, an expression e, and the revision σ. For a server identifier s, the server state $\mathcal{C}(s)$ is a tuple (r, R, σ) consisting of a round number r, a round map R and a revision state σ.

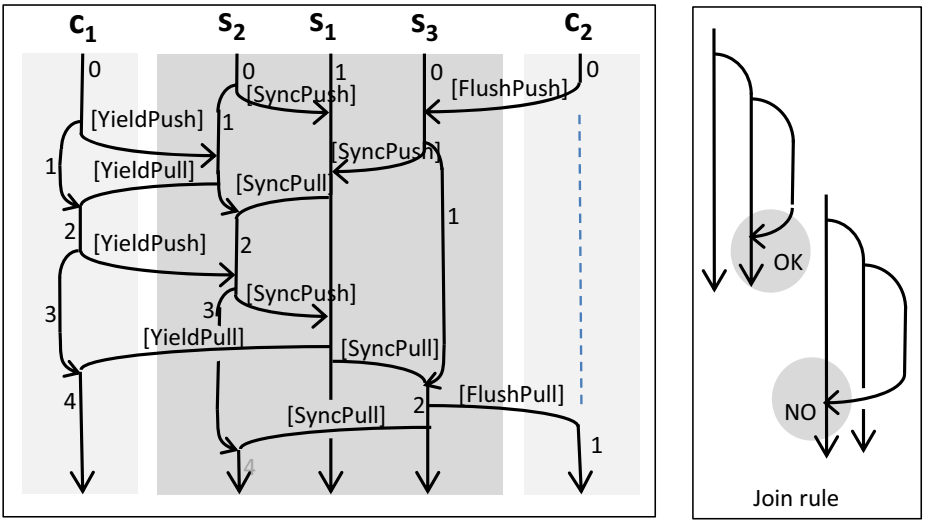


Fig. 8. **(a)** (left) An illustration of an execution with 2 clients and 3 servers. **(b)** (right) An illustration of the join rule.

The revision state σ represents the state of the current replica. We defer the description of the implementation of σ until Section 5, where we discuss cloud types and define fork-join automata. For now, we simply postulate that σ is in some set Σ, supports all the local data operations, has an initial state σ_0, and supports fork and join functions $fork : \Sigma \to \Sigma \times \Sigma$ and $join : \Sigma \times \Sigma \to \Sigma$, respectively. Moreover, we assume that $fork(\sigma_0) = (\sigma_0, \sigma_0)$ (forking from the initial state yields the initial state).

The round numbers r are used to track which clients (and servers) can synchronize with particular servers. After each fork, the round number of a client or server is incremented. The round map R on a server s is a total function that maps each identifier i of a client or server to a round number $R(i)$ which is the number of the last round whose fork is in the visible history of s. The initial round map R_0 maps all clients and servers to round 0 (since round 0 is always forked from the initial state of the main revision, it is retroactively in the visible history). The rules are set up to enforce that a client c (or server s) with round number r can only communicate with a server where $R(c) = r$.

Fig. 9 presents transition rules of the form $\mathcal{C} \Rightarrow \mathcal{C}'$ where cloud state $\mathcal{C}$ updates to $\mathcal{C}'$. We use the pattern match notation $\mathcal{C}(a_1 \mapsto b_1, ..., a_n \mapsto b_n)$ to match on a partial function $\mathcal{C}$ satisfying $\mathcal{C}(a_i) = b_i \ \forall i.1 \leqslant i \leqslant n$. We write $\mathcal{C}[a \mapsto b]$ to denote a partial function that is equivalent to $\mathcal{C}$ except that $\mathcal{C}(a) = b$.

For any cloud state there are potentially many valid transitions which capture the inherent concurrency and non-determinism of cloud execution. For example, clients can be spawned at any time using the rule [Spawn],and clients can arbitrarily interleave local evaluation.

The rules [Yield-Nop], [Yield-Push] and [Yield-Pull] describe how clients synchronize with servers. The [Yield-Nop] states that a *yield* instruction can be ignored,

$$[\textsc{Eval}]\ \frac{e;\sigma \to e';\sigma'}{\mathcal{C}(c \mapsto (r,e,\sigma)) \ \Rightarrow\ \mathcal{C}[c \mapsto (r,e',\sigma')]}$$

$$[\textsc{Spawn}]\ \frac{c \notin \mathsf{dom}(\mathcal{C})}{\mathcal{C} \ \Rightarrow\ \mathcal{C}[c \mapsto (0,e,\sigma_0)]}$$

$$[\textsc{Yield-Push}]\ \frac{R(c)=r \quad R'=R[c\mapsto r+1] \quad \mathit{fork}(\sigma_c)=(\sigma'_c,\sigma''_c) \quad \mathit{join}(\sigma_s,\sigma'_c)=\sigma'_s}{\mathcal{C}(s\mapsto(r_s,R,\sigma_s),\ c\mapsto(r,\mathcal{E}[\![\mathit{yield}]\!],\sigma_c)) \ \Rightarrow\ \mathcal{C}[s\mapsto(r_s,R',\sigma'_s),\ c\mapsto(r+1,\mathcal{E}[\![()]\!],\sigma''_c)]}$$

$$[\textsc{Yield-Pull}]\ \frac{R(c)=r \quad R'=R[c\mapsto r+1] \quad \mathit{fork}(\sigma_s)=(\sigma'_s,\sigma''_s) \quad \mathit{join}(\sigma''_s,\sigma_c)=\sigma'_c}{\mathcal{C}(s\mapsto(r_s,R,\sigma_s),\ c\mapsto(r,\mathcal{E}[\![\mathit{yield}]\!],\sigma_c)) \ \Rightarrow\ \mathcal{C}[s\mapsto(r_s,R',\sigma'_s),\ c\mapsto(r+1,\mathcal{E}[\![()]\!],\sigma'_c)]}$$

$$[\textsc{Yield-Nop}]\ \frac{}{\mathcal{C}(c\mapsto(r,\mathcal{E}[\![\mathit{yield}]\!],\sigma)) \ \Rightarrow\ \mathcal{C}[c\mapsto(r,\mathcal{E}[\![()]\!],\sigma)]}$$

Fig. 9. Cloud evaluation rules for clients

$$[\textsc{Create}]\ \frac{s \notin \mathsf{dom}(\mathcal{C})}{\mathcal{C} \ \Rightarrow\ \mathcal{C}[s\mapsto(0,R_0,\sigma_0)]}$$

$$[\textsc{Sync-Push}]\ \frac{R_s(t)=r_t \quad R'_s=\mathsf{max}(R_s,R_t) \quad R''_s=R'_s[t\mapsto r_t+1] \quad \mathit{fork}(\sigma_t)=(\sigma'_t,\sigma''_t) \quad \mathit{join}(\sigma_s,\sigma'_t)=\sigma'_s}{\mathcal{C}(s\mapsto(r_s,R_s,\sigma_s),\ t\mapsto(r_t,R_t,\sigma_t)) \ \Rightarrow\ \mathcal{C}[s\mapsto(r_s,R''_s,\sigma'_s),\ t\mapsto(r_t+1,R_t,\sigma''_t)]}$$

$$[\textsc{Sync-Pull}]\ \frac{R_s(t)=r_t \quad R'_t=\mathsf{max}(R_s,R_t) \quad R'_s=R_s[t\mapsto r_t+1] \quad \mathit{fork}(\sigma_s)=(\sigma'_s,\sigma''_s) \quad \mathit{join}(\sigma''_s,\sigma_t)=\sigma'_t}{\mathcal{C}(s\mapsto(r_s,R_s,\sigma_s),\ t\mapsto(r_t,R_t,\sigma_t)) \ \Rightarrow\ \mathcal{C}[s\mapsto(r_s,R'_s,\sigma'_s),\ t\mapsto(r_t+1,R'_t,\sigma'_t)]}$$

$$[\textsc{Retire}]\ \frac{R_s(t)=r_t \quad R'_s=\mathsf{max}(R_s,R_t) \quad \mathit{join}(\sigma_s,\sigma_t)=\sigma'_s}{\mathcal{C}(s\mapsto(r_s,R_s,\sigma_s),\ t\mapsto(r_t,R_t,\sigma_t)) \ \Rightarrow\ \mathcal{C}[s\mapsto(r_s,R'_s,\sigma'_s),\ t\mapsto\bot]}$$

Fig. 10. Cloud evaluation rules for servers

allowing disconnected clients to keep executing. The rule [Yield-Push] sends a revision to an eligible server, while the rule [Yield-Pull] receives a revision from an eligible server. In both cases, the round number of the client is incremented and the round map of the server is updated. The new states of the client and server are determined by forking/joining revisions appropriately (see Fig. 8).

Figure 10 shows the rules for server synchronization. The rules [Create] and [Retire] create and retire servers on demand. The [Sync] rule is the synchronization rule for servers and is like a simplified (more synchronous) version of [Yield]. The premise ensures that the round number matches, the round number is incremented, and the state is first joined and then forked again. What is different is that the round maps of both servers are also joined using $R = \mathsf{max}(R_s, R_t)$ (taking the pointwise max of the vector clocks).

$$[\textsc{Flush-Push}]\quad \frac{R(c) = r \qquad R' = R[c \mapsto r+1] \qquad join(\sigma_s, \sigma_c) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\textit{flush}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R', \sigma'_s),\ c \mapsto (r+1, \mathcal{E}[\![\textit{block}]\!], \sigma_c)]}$$

$$[\textsc{Flush-Pull}]\quad \frac{R(c^{\mathsf{flush}}) = r \qquad fork(\sigma_s) = (\sigma'_s, \sigma'_c)}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\textit{block}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R, \sigma'_s),\ c \mapsto (r, \mathcal{E}[\![()]\!], \sigma'_c)]}$$

$$[\textsc{Commit}]\quad \frac{R' = R[\forall c.\ c^{\mathsf{flush}} \mapsto R(c))}{\mathcal{C}(s_{\mathsf{main}} \mapsto (0, R, \sigma)) \ \Rightarrow\ \mathcal{C}[s_{\mathsf{main}} \mapsto (0, R', \sigma)]}$$

Fig. 11. Semantics of the *flush* operation

4.1 Flush

To describe the *flush* operation, we distinguish an initial main server s_{main}. The *flush* operation must not only guarantee that all updates of a client are joined in the main server s_{main}, but also that the client sees all the state changes in the main server that were applied before the client state was joined.

To track which client updates have been seen by the main server, we add an extra round number c^{flush} in the round map. As shown in Figure 11, the main server can always execute the rule [Commit] to set the c^{flush} entries to the corresponding round numbers of the clients that have synchronized with the main server. Through rule [Sync] (Fig 10) any servers that synchronize with the main server will propagate these c^{flush} entries automatically.

The rule [Flush-Push] is applied whenever the client does a *flush* operation. The rule is similar to [Yield-Push] but blocks the client. Also, only the state of the client is joined with the server state, but the client state itself does not fork a new revision. The round map of the server s is updated though with the new round number $c \mapsto r+1$. Now, servers can execute [Sync] until the state changes are propagated all the way up to s_{main}. At that point, the main server can make a [Commit] transition, making $c^{\mathsf{flush}} \mapsto r{+}1$. After again doing more [Sync] transitions, the new c^{flush} entry makes it back to the original server. At this point, [Flush-Pull] can apply where the server state is forked now into a new server state σ'_s and client state σ'_c, and where the client is unblocked again.

4.2 Message Protocols and Server State

The rules presented are still somewhat more abstract than needed for an actual implementation, to keep the presentation from becoming too technical. In practice, all communication is asynchronous (based on message delivery) and unreliable. Thus, our actual implementation breaks synchronous transition rules (like Yield-Push, Yield-Pull, Sync, Flush-Pull, Flush-Push) into message protocols, uses state machines that are locally persisted, and retransmits messages if they are lost.

Another very important optimization concerns the size of messages. Sending the full replica state in messages is of course impractical. Thus we use compression by sending diffs of the state.

5 Cloud Types

We now examine our cloud type implementations in more detail. To this end we define the concept of a *fork-join automaton*. Fork-join automata are concrete implementations of cloud types, consisting of implementations for the abstract update and query operations, and concrete implementations of fork and join.

Definition 1. *A* fork-join automaton *(FJA) is a tuple* $(Q, U, \Sigma, \sigma_0, f, j)$ *where*

- Q *is an abstract set of query operations*
- U *is an abstract set of update operations*
- Σ *is a set of states*
- $\sigma_0 \in \Sigma$ *is the initial state*
- *Queries and updates have an interpretation as functions, specifically (1) each query operation* $q \in Q$ *defines a function* $q^\# : \Sigma \to Val$*, and (2) each update operation* $u \in U$ *defines a function* $u^\# : \Sigma \to \Sigma$*.*
- $f : \Sigma \to \Sigma \times \Sigma$ *is a function for splitting the current state on a fork.*
- $j : \Sigma \times \Sigma \to \Sigma$ *is a function for merging states on a join.*

Fork-join automata must satisfy a correctness conditions: they must correctly track and apply updates when revisions are forked and joined (as we illustrated earlier in Section 2.2). We discuss this condition only informally here, since its definition depends on the definition of revision diagrams, which is outside the scope of this paper. A full exposition is available in [1].

In the remainder of this section, we define a fork-join automaton for the entire cloud state (i.e. for all cloud data declared by the user). First, we define fork-join automata for the primitive cloud types **CInt** and **CString**. Then we show how to define the cloud types for entities and arrays. Finally, we show how to provide the cloud type **CSet** as syntactic sugar.

5.1 A Fork-Join Automaton for CInt

For cloud integers, we support operations get and set to read and write the current value, as well as add (Fig. 12). In the state, we store three values: a boolean indicating whether the current revision performed any set operations, a base value, and an offset. On fork, the boolean is reset, the base value is set to the current value, and the offset is set to zero. Add operations change only the offset, while set operations set the boolean to true, set the base value, and reset the offset. On join, we assume the value of the joined revision (if it performed a set) or add its offset (otherwise). This produces the desired semantics (see Section 2.2 for examples).

$$
\begin{aligned}
Q^{\mathbf{CInt}} &: \{\mathsf{get}\} \\
U^{\mathbf{CInt}} &: \{\mathsf{set}(n) \mid n \in \mathrm{int}\} \cup \{\mathsf{add}(n) \mid n \in \mathrm{int}\} \\
\Sigma^{\mathbf{CInt}} &: \mathrm{bool} \times \mathrm{int} \times \mathrm{int} \\
\sigma_0^{\mathbf{CInt}} &: (\mathrm{false}, 0, 0) \\
\mathsf{add}(n)^{\#}\ (r,b,d) &= (r, b, d+n) \\
\mathsf{set}(n)^{\#}\ (r,b,d) &= (\mathrm{true}, n, 0) \\
\mathsf{get}^{\#}\ (r,b,d) &= b + d \\
f^{\mathbf{CInt}}\ (r,b,d) &= (r,b,d), (\mathrm{false}, b+d, 0) \\
j^{\mathbf{CInt}}\ (r_1,b_1,d_1)(r_2,b_2,d_2) &= \begin{cases} (\mathrm{true}, b_2, d_2) & \text{if } r_2 = \mathrm{true} \\ (r_1, b_1, d_1 + d_2) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 12. Fork-join automaton for **CInt**

5.2 A Fork-Join Automaton for CString

For cloud strings, we support operations get and set to read and write the current value, and a conditonal operation setIfEmpty (Fig. 13). In the state, we record the current value and whether it has not been written ($\bot$), has been written (wr), or has been conditionally written (cond). A conditional write succeeds only if the current value is empty, and this test is repeated on merge.

5.3 A Fork-Join Automata for the Complete State

For a fixed schema $\mathcal{S}$, we can now define the entire state as a fork-join automaton. First, we define the query operations $Q^{\mathcal{S}}$ and the update operations $U^{\mathcal{S}}$ as in the following table.

$$
\begin{aligned}
Q^{\mathbf{CString}} &: \{\mathsf{get}\} \\
U^{\mathbf{CString}} &: \{\mathsf{set}(s) \mid s \in \mathrm{string}\} \cup \{\mathsf{setIfEmpty}(s) \mid s \in \mathrm{string} \setminus \{\text{""}\}\} \\
\Sigma^{\mathbf{CString}} &: \{\bot, \mathsf{wr}, \mathsf{cond}(\mathrm{string})\} \times \mathrm{string} \\
\sigma_0^{\mathbf{CString}} &: (\bot, \text{""}) \\
\mathsf{set}(s)^{\#}\ (r,t) &= (\mathsf{wr}, s) \\
\mathsf{setIfEmpty}(s)^{\#}\ (r,t) &= \begin{cases} (\mathsf{wr}, s) & \text{if } r = \mathsf{wr} \wedge t = \text{""} \\ (\mathsf{cond}(\mathrm{s}), s) & \text{if } r = \bot \wedge t = \text{""} \\ (\mathsf{cond}(\mathrm{s}), t) & \text{if } r = \bot \wedge t \neq \text{""} \\ (r,t) & \text{otherwise} \end{cases} \\
\mathsf{get}^{\#}\ (r,s) &= s \\
f^{\mathbf{CString}}\ (r,s) &= (r,s), (\bot, s) \\
j^{\mathbf{CString}}\ (r_1,s_1)(r_2,s_2) &= \begin{cases} (\mathsf{wr}, s_2) & \text{if } r_2 = \mathsf{wr} \\ (\mathsf{wr}, s) & \text{if } r_1 = \mathsf{wr} \wedge s_1 = \text{""} \wedge r_2 = \mathsf{cond}(s) \\ (\mathsf{cond}(s), s) & \text{if } r_1 = \bot \wedge s_1 = \text{""} \wedge r_2 = \mathsf{cond}(s) \\ (\mathsf{cond}(s), s_1) & \text{if } r_1 = \bot \wedge s_1 \neq \text{""} \wedge r_2 = \mathsf{cond}(s) \\ (r_1, s_1) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 13. Fork-join automaton for **CString**

operation	argument types	return type	entity/property definition
all_E		$\mathbf{Set}\langle E\rangle$	*entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{create}_E(e)$	E		*entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{delete}_E(e)$	E		*entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{entries}_p$		$\mathbf{Set}\langle\iota\rangle$	*property* $p \ : \ \iota \to \omega$
$\mathsf{query}_p(i, q)$	ι, Q^ω	Val	*property* $p \ : \ \iota \to \omega$
$\mathsf{update}_p(i, u)$	ι, U^ω		*property* $p \ : \ \iota \to \omega$

Next, we define the state space to consist of separate components for each entity type and each property

$$\Sigma^{\mathcal{S}} = \prod_{p \in \mathcal{S}} \Sigma_p \times \prod_{E \in \mathcal{S}} \Sigma_E.$$

For each declaration *property* $p \ : \ \iota \to \omega$ we store a total function from keys to values, where keys are of the corresponding index type, and values belong to the state space of the corresponding fork-join automaton:

$$\Sigma_p = \iota \to \Sigma^\omega$$

For each declaration *entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$ we store a total function from entities to a state that indicates whether this entity is not yet created ($\bot$), exists as a normal entity (ok), or has been deleted ($\top$):

$$\Sigma_E = E \to \{\bot, ok, \top\}$$

For a state $\sigma \in \Sigma^{\mathcal{S}}$, we let σ_p and σ_E be the projection on the respective components.

Naturally, in the initial state $\sigma_0^{\mathcal{S}}$, we map all property indexes to Σ_0^ω (the initial state of the corresponding fork-join automaton) and all entities to $\bot$. We show the implementation of queries, updates, fork, and join in Fig. 14, using pseudocode, and explain them in the following.

- Create adds a fresh element to an entity by mapping it to ok. We assume each client can create fresh elements (based on a local id and counter).
- Delete maps the deleted element to $\top$ to mark it as deleted. We cannot simply remove it because at joins, it would be impossible to determine if one side is fresh, or the other deleted. Extra book-keeping can be used to eventually collect these tombstones.
  Deletion also causes any dependent entities to be deleted. This is achieved by Propagate. Note that entity dependencies cannot be cyclic, since an entity can only be used in the creation of another when it is already defined.
- all_E returns all non-deleted values of a given entity.
- A query q on an entry i of property p is answered by delegating it to the FJA of p at i, provided that i is not deleted.
- Similarly, an update u on an entry i of property p is delegated to the FJA of p at i, provided i is not deleted.
- $\mathsf{entries}_p$ returns all the entries of a property p that map to non-default FJAs and are not deleted.

- Forking the overall FJA turns into a point-wise forking of all the FJA's of each property. The entity maps are unaffected by forking.
- Joining is similarly performed point-wise on all properties. For entities, joining requires computing the maximum in the order $\bot < ok < \top$. This achieves deleting the entry, provided any one side has it deleted, or keeping it allocated, if any one side has it allocated. At joins, we also need to repropagate deletions to all dependent elements, as new deletions can be merged into the revision.

It is remarkable that the complete state FJA operations are commutative by themselves. The only non-commutative operations are in the FJAs implementing cloud types. This property makes using arrays and entities very natural and does not introduce unexpected conflict resolutions. Furthermore, our design was careful to enable a completely modular implementation of the complete state FJA with respect to the cloud type implementations. In part, this structure makes a single parameterized, reusable implementation of cloud storage and synchronization possible. Any schema and any extensions of cloud types can be supported without further changes.

5.4 Implementation of CSet

Rather than defining sets directly, we encode them relationally, building on the abstraction mechanism provided by entities. Given a schema definition for a property of type **CSet**$\langle\iota\rangle$, we rewrite it to an entity definition whose entities represent "instances" of additions of elements:

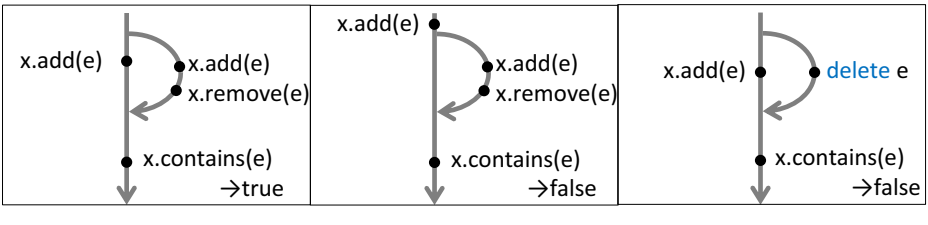
property p: $\iota' \to$ **CSet**$\langle\iota\rangle$ $\equiv$ *entity* E_p[index : ι', element : ι]

Then we encode operations as follows:

x.add(i) $\equiv$ { *new* $E_p(x, i)$; }
x.contains(i) $\equiv$ { *return* (*all* E_p *where* index == x and element == i).isNotEmpty() ; }
x.remove(i) $\equiv$ { *foreach* (e *in all* E_p *where* index == x and element == i) e.*delete*() ; }

Our indirect encoding has two advantages (illustrated in the picture below):

- Removing an element from a set only removes instances that were visibly added before the remove. This is known as observed-remove behavior, as proposed in [13] as a reasonable semantics for eventually consistent sets (see examples on the left and in the middle below).
- If the user deletes an entity, that entity disappears automatically from all sets that contain it (see example on the right below).



```
// operations on entities

create_E(e) {
    σ_E(e) := σ_E(e)[e ↦ ok];
}
delete_E(e) {
    σ_E(e) := σ_E(e)[e ↦ ⊤];
    propagate();
}
all_E {
    return {e ∈ E | σ_E(e) = ok};
}

// auxiliary functions

propagate() {
    while exists E, e such that
      σ_E(e) ≠ ⊤ and deleted(e)
    do
      σ_E(e) := σ_E(e)[e ↦ ⊤];
}
deleted(i) {
    match i with
    A[i_1, ..., i_n]:
        return (exists j such that deleted(i_j));
    E[uid, i_1, ..., i_n]:
        return σ_E(i) = ⊤
            or (exists j such that deleted(i_j));
    else // string or int
        return false;
}
isdefault(σ) {
    if σ ∈ Σ^CInt
        return get^#σ = 0;
    else if σ ∈ Σ^CString
        return get^#σ = "";
    else if σ ∈ Σ^CSet⟨ι⟩
        return elems^#σ = ∅;
}
```

```
// operations on properties

query_p(i, q) {
    if (deleted(i))
      return ⊥;
    else
      return σ_p(i).q;
}
update_p(i, u) {
    if (not deleted(i))
      σ_p(i).u;
}
entries_p {
    return all i ∈ ι
      where (not isdefault(σ_p(i))
        and (not deleted(i))
}


// fork and join functions

fork() {
    var σ' = σ; // copy the state
    foreach property p : ι → ω
      foreach i ∈ ι
        (σ_p(i), σ'_p(i)) := f^ω(σ_p(i));
    return σ';
}

join(σ') {
    foreach property p : ι → ω
      foreach i ∈ ι
        σ_p(i) := j^ω(σ_p(i), σ'_p(i));
    foreach entity E(k_1 : ι_1, ..., k_n : ι_n)
      foreach e ∈ E
        σ_E(e) := max(σ_E(e), σ'_E(e)));
    propagate();
}

max(s_1, s_2) uses the order ⊥ < ok < ⊤
```

Fig. 14. Complete fork-join automaton

6 Related Work

At the heart of our work is the idea of using revision diagrams and fork-join automata to achieve eventual consistency, which was introduced in [1]. In this paper we extend and concretize this idea, by (1) devising a composable way to

construct schema from basic cloud types, which eliminates the need for user-defined conflict resolution code, (2) giving examples of concrete programs and cloud types, (3) devising primitives that are sufficient to recover stronger synchronization. Moreover, we provide a formal syntax and semantics that connects a small, but sufficiently expressive programming language with a detailed operational system model.

Eventual consistency is motivated by the impossibility of achieving strong consistency, availability, and partition tolerance at the same time, as stated by the CAP theorem [5]. Eventual consistency across the literature uses a variety of techniques to propagate updates (e.g. general causally-ordered broadcast [14,15], or pairwise anti-entropy [10]). For a general high-level comparison of our work with various notions of eventual consistency appearing in the literature, we refer to the discussion in [1].

Most closely related to our work are conflict-free replicated data types (CRDTs) [14] and Bayou's weakly consistent replication [16].

- CRDTs are very similar to our cloud types, insofar that they separate the use of eventually consistent data types from their implementation. In fact, CRDTs can serve as cloud types (as exemplified by the observed-remove set proposed in [13]). However, we are not aware of prior work on how to compose individual CRDTs into a larger schema. Furthermore, CRDTs only support commutative operations, whereas our approach supports non-commutative operations while still achieving eventual consistency. Furthermore, we support stronger synchronization primitives like *flush* when necessary, in the same framework.
- In Bayou [16], and in the original Concurrent Revisions work[2], conflict resolution is achieved by explicit merge functions written by the user. In contrast, this paper uses conflict resolution that is *automatically inferred* from the structure of the type declarations.

Research on *persistent data types* [8] is related to our definition of cloud types insofar it concerns itself with efficient implementations of data types that permit retrieval and mutations of past versions. However, it does not concern itself with apects related to transactions or distribution.

Prior work on *operational transformations* [15] can be understood as a specialized form of eventual consistency where updates are applied to different replicas in different orders, and modified in such a way as to guarantee convergence. This specialized formulation can provide highly efficient broadcast-based real-time collaboration, but poses significant implementation challenges [7].

There is of course a large body of work on transactions. Most academic work considers strong consistency (serializable transactions) only, and is thus not directly applicable to eventual consistency. Nevertheless there are some similarities, such as:

- [6] provides insight on the limitations of serializable transactions, and proposes similar workarounds as used by eventual consistency (timestamps and commutative updates). However, transactions remain tentative during disconnection.

- Snapshot isolation [4] relaxes the consistency model, but transactions can still fail, and can not commit in the presence of network partitions.
- Automatic Mutual Exclusion [9], like our work, uses *yield* statements to separate transactions.

7 Conclusion

Providing good programming abstractions for cloud storage, synchronization, and disconnected operation appears crucial to accelerate the production of useful and novel applications on today's and tomorrow's mobile devices. In this paper, we provided a sound foundation upon which to build such programming abstractions through the use of automatically synchronized cloud data types that can be composed into a larger data schema using indexed arrays and entities. The design we presented allows implementing all the difficult parts of such a system (the cloud service, the local persistence, the caching, the conflict resolution, and the synchronization) once and for all, while guaranteeing eventual consistency. An application programmer declares only the data schemas and focuses on writing code performing operations on the data, as well as identifying points in his program where synchronization is desired.

References

1. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually Consistent Transactions. In: Seidl, H. (ed.) Programming Languages and Systems. LNCS, vol. 7211, pp. 67–86. Springer, Heidelberg (2012)
2. Burckhardt, S., Leijen, D.: Semantics of Concurrent Revisions. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 116–135. Springer, Heidelberg (2011)
3. Chen, P.P.-S.: The entity-relationship model toward a unified view of data. ACM Trans. Database Syst. 1, 9–36 (1976)
4. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2), 492–528 (2005)
5. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 51–59 (2002)
6. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. SIGMOD Record 25, 173–182 (1996)
7. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. Theoretical Computer Science 351, 167–183 (2006)
8. Kaplan, H.: Persistent data structures. In: Handbook on Data Structures and Applications, pp. 241–246. CRC Press (1995)
9. Martin, A., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: Principles of Programming Languages (POPL), pp. 63–74 (2008)
10. Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Demers, A.: Flexible update propagation for weakly consistent replication. Operating Systems Review 31, 288–301 (1997)

11. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys 37, 42–81 (2005)
12. Shapiro, M., Kemme, B.: Eventual consistency. In: Encyclopedia of Database Systems, pp. 1071–1072. Springer (2009)
13. Shapiro, M., Preguica, N., Baquero, C., Zawirski, M.: A comprehensive study of convergent and commutative replicated data types. Technical Report Rapport de recherche 7506, INRIA (2011)
14. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-Free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011)
15. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Conference on Computer Supported Cooperative Work, pp. 59–68 (1998)
16. Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing update conflicts in bayou, a weakly connected replicated storage system. SIGOPS Oper. Syst. Rev. 29, 172–182 (1995)
17. Tillmann, N., Moskal, M., de Halleux, J., Fähndrich, M.: Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In: ONWARD 2011 at SPLASH (also available as Microsoft TechReport MSR-TR-2011-49) (2011)

Lock Inference in the Presence of Large Libraries

Khilan Gudka[1], Tim Harris[2], and Susan Eisenbach[1]

[1] Imperial College London
{khilan,susan}@imperial.ac.uk
[2] Microsoft Research Cambridge
tharris@microsoft.com

Abstract. Atomic sections can be implemented using lock inference. For lock inference to be practically useful, it is crucial that large libraries be analysed. However, libraries are challenging for static analysis, due to their cyclomatic complexity.

Existing approaches either ignore libraries, require library implementers to annotate which locks to take or only consider accesses performed upto one level deep in library call chains. Thus, some library accesses may go unprotected, leading to atomicity violations that atomic sections are supposed to eliminate.

We present a lock inference approach for Java that analyses library methods in full. We achieve this by (i) formulating lock inference as an Interprocedural Distributive Environment dataflow problem, (ii) using a graph representation for summary information and (iii) applying a number of optimisations to our implementation to reduce space-time requirements and locks inferred. We demonstrate the scalability of our approach by analysing the entire GNU Classpath library comprising 122KLOC.

1 Introduction

Atomic sections [1] are an abstraction for shared-memory concurrency. They allow a programmer to demarcate a block of code that should execute without interference from concurrent threads but leave the low-level details of achieving this to the compiler and/or run-time. If used correctly, they can remove many of the problems that have plagued programmers for decades, such as low-level race conditions, deadlock, priority inversion and convoying [2]. With current abstractions, the frequency of such unfortunate encounters is only likely to increase, given that multi-core processors are the norm [3, 4].

Atomic sections are a language-level construct, hence an important question is, *how should we implement them?* Software Transactional Memory (STM) [5] is a popular approach, wherein memory updates are buffered during execution and then committed atomically. If a conflicting update has already been committed by another thread, the buffer is discarded and the transaction is re-executed (*rollback*). This ability to abort and re-run is essential to STM, as implementations are typically *optimistic*; they execute with the assumption that interference is unlikely to occur. Rolling back execution is unappealing because irreversible

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 308–332, 2012.

operations (e.g. system calls) cannot be rolled back and performance can be harmed by maintaining undo-logs to allow rollback.

In light of these shortcomings, pessimistic alternatives have been proposed, based on *lock inference*. These alternatives statically infer enough locks to prevent interference and instrument the program with the corresponding lock operations. Since lock inference must consider all possible execution paths, this compile-time approach may introduce more lock/unlock operations than strictly necessary, resulting in less concurrency.

Real-world programs make extensive use of libraries, hence being able to analyse them is important. However, libraries create a scalability challenge for static analysis [6] because they are large and have a high cyclomatic complexity.[1] Most problematic is that an analysis may not be able to complete if the memory requirements are too great. Furthermore, even simple programs can involve vast amounts of library code. Consider a "Hello World!" example extended with atomic sections:

```
atomic {
    System.out.println("Hello World!");
}
```

Lock inference prides itself on being able to support I/O, so one would expect it to be able to handle this library call. In practice, this example is non-trivial with a compile-time call graph containing 1150 library methods for GNU Classpath 0.97.2. Analysing the library is a hard problem as is evident from the fact that existing work either ignores libraries [8–11], requires library implementers to annotate which method parameters should be locked prior to the call [12] or only considers accesses performed upto one level deep in library call chains [13]. All of these have the potential that some shared accesses performed within the library may go unprotected, leading to atomicity violations. Our previous approach [14] on this example was intractable.

Our main contribution is a lock inference approach for Java that analyses library methods in full. Specifically:

- We formulate lock inference as an Interprocedural Distributive Environment (IDE) dataflow problem.
- We adapt the pointwise graph representation of Sagiv et al [15] to reduce the number of edges in our summary graphs.
- We present *delta transformers* that dramatically reduce IDE analysis space-time requirements by only propagating new dataflow information.
- We identify and remove many locks for thread-local, internal, dominated and read-only objects.
- We implement our whole-program analyses in Soot.

[1] Cyclomatic complexity [7] is a measure of the number of linearly independent paths. Library call chains can be long and consist of large strongly connected components.

We evaluate our approach as follows:

- We demonstrate analysis scalability by analysing the entire GNU Classpath library (122KLOC) and the popular Java SQL database engine HSQLDB (150KLOC) on top of GNU Classpath.
- We evaluate the effects of a number of analysis optimisations: delta transformers, CFG summarisation [6], parallel processing of worklists and worklist ordering. We show that our delta transformers give the biggest speedup whilst also reducing memory usage.
- We evaluate the run-time performance of a range of benchmarks instrumented with our locks and compare results with the original synchronisation and Halpert et al [13].

2 General Approach

Our general approach is to use the Soot framework [16] to analyse Java classes annotated with atomic sections (we treat synchronized blocks and methods as atomic sections) and replace these annotations with suitable locks. Our analysis ensures weak atomicity.

First, we perform a dataflow analysis to infer what objects are accessed in each atomic section. Nested atomics are flattened and merged. We compute a summary for each method, which describes the accesses performed by it and all transitively called methods. The result is a graph describing all objects accessed in the atomic section, which we convert to locks.

We infer *instance locks* where possible, however, for those portions of the graph that describe a statically unbounded set of accesses, we infer locks on the *types* of these objects. We use *multi-granularity locking* [17] to support both

```
1 class Scheduler {
2     Printer p1, p2;
3
4     atomic boolean schedule(Job j) {
5         // lockRead(this)
6         // lockWrite(this.p1);
7         // lockWrite(this.p2);
8         if (this.p1.job == null) {
9             this.p1.job = j;
10        } else if (this.p2.job == null) {
11            this.p2.job = j;
12        }
13        // unlockWrite(this.p2);
14        // unlockWrite(this.p1);
15        // unlockRead(this)
16    }
17 }
```

Fig. 1. An example atomic method and the locks we infer

kinds of lock simultaneously: a type lock can be acquired if none of the locks on its instances are currently acquired and vice-versa.

We use simple analyses to identify objects that don't have to be locked such as thread-local or internal objects. We also detect when only a single thread is executing to avoid acquiring/releasing locks.

Finally, we instrument the program with the inferred set of locks, such that they are acquired upon entry to the atomic section and released upon exit. Acquiring all locks together at the start, allows us to test for deadlock at run-time. If it occurs, we release all locks that have already been acquired and subsequently attempt to re-acquire them. As no updates have been performed this is safe.

Fig. 1 shows an atomic method and the lock operations that would be instrumented by our analysis. The example consists of two `Printer`s and a `Scheduler`, which allocates a given job to the next available `Printer` (which handles one job at a time). Statically, we can't be sure which conditional branch will be executed, so we must acquire a write lock on both `Printers`.

3 Inferring Accesses

In this section, we present our analysis for inferring which objects are accessed by each atomic section. We represent object accesses as syntactic expressions called *paths* [18, 14]. A path is an expression used to identify an object in code and consists of a variable followed by zero or more field and/or array lookups in any order. An example of a path expression is `x.f.g[i].h`. Our main contribution is that our analysis can scale to large programs that make use of large libraries.

We compute a summary function for each method m that describes the cumulative effects of m (including all methods transitively called by m). These functions are computed by composing the individual transfer functions for each of m's statements where the dataflow information are these transfer functions. For scalability, it is essential to have a compact representation for transfer functions with fast composition and meet operations.

For the general class of dataflow problems, called *Interprocedural Distributive Environment* (IDE), Sagiv et al [15] represent transfer functions as graphs, allowing composition to be computed by taking the transitive closure and meet by graph union or intersection. Rountev et al [6] have also shown that IDE analyses with this representation can scale well to programs using large Java libraries. We thus formulate our analysis as an IDE dataflow problem. In an IDE problem, dataflow values are mappings called *environments*. Transfer functions are called *environment transformers*.

In previous work [14], we represent sets of paths as *non-deterministic finite automata* (NFA). Fig. 2(a) shows an example NFA we might infer for the set {`this`, `this.pl`}. In our NFAs, numbers within states refer to the CFG node that generated the access. So, in this case, CFG nodes 1 and 2 generated an access of `this` and CFG node 3 generated an access of `this.p1` (whereby the `this` was generated at CFG node 1). We assume three-address code and thus each CFG node dereferences at-most one object. Labelling NFA states with the CFG

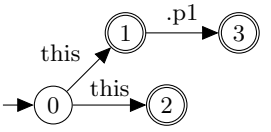


Fig. 2. (a) Example NFA for the set of paths $\{this, this.p1\}$ and its environment representation (b)

node that generated them allows us to efficiently detect looping accesses [14]. Transition labels correspond to variables, class names (for static lookups), field names and `[*]` to represent an array lookup.

IDE analyses require dataflow values to be maps, so in this paper, we represent NFAs as mappings of the form $\Sigma \to \mathcal{P}(Q \times Q)$, where Σ is the set of transition labels and Q is the set of NFA states. The states in each pair refer to the source and destination of the transition respectively.

3.1 IDE Transformers

We now define the environment transformers for our analysis. Transformers describe how dataflow values, i.e. environments, should be translated for a particular program statement. The challenge we face is that the object referred to by a path, such as `x`, may differ between the point where `x` is dereferenced and the point where locks are acquired, due to assignments that occur in-between. Our analysis is a backwards analysis because we push path expressions upwards. Our transformers translate these paths to preserve the set of objects that are accessed below, albeit potentially introducing new accesses due to the conservatism of our alias analysis (we use type information).

Fig. 3 contains our transformers, which we now describe in turn. We use Soot's three-address *Jimple representation*. We also assume a control flow graph (CFG) exists, whereby each CFG node is labelled with a unique identifier n. We represent a CFG node in text with the notation $[...]^n$

$[x = y]^n$. The object referenced by `x` after this assignment was pointed-to by `y` before the assignment. To preserve object accesses performed lower down, paths beginning with `x` are rewritten to begin with `y`. We achieve this by modifying the incoming environment e by replacing all automaton transitions of the form $0 \xrightarrow{x} n'$ with $0 \xrightarrow{y} n'$. This involves copying `x`'s transitions to `y`'s set: $\texttt{y} \mapsto e(y) \cup e(x)$, and deleting `x`'s transitions: $\texttt{x} \mapsto \emptyset$.

$[\texttt{x = new}]^n$ *and* $[\texttt{x = null}]^n$. In these two cases, accesses of `x` below the assignment will either be local to the atomic section (`new`) or generate a `NullPointerException` (`null`). No locks need to be acquired, so we delete paths beginning with `x` by removing all $0 \xrightarrow{x} n'$ transitions: $\texttt{x} \mapsto \emptyset$.

$$
\begin{array}{rcl}
t_{[\texttt{x = y}]^n} & = & \lambda e.e[y \mapsto e(y) \cup e(x)][x \mapsto \emptyset] \\
t_{[x=\texttt{null or new}]^n} & = & \lambda e.e[x \mapsto \emptyset] \\
\hline
t_{[x=y.f]^n} & = & \lambda e.e[y \mapsto e(y) \cup \{(0,n)\}] \\
 & & \quad [.f \mapsto e(.f) \cup \{(n,n') | (0,n') \in e(x)\}] \\
 & & \quad [x \mapsto \emptyset] \\
\hline
t_{[x.f=y]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
 & & \quad [y \mapsto e(y) \cup \{(0,n'') | (n',n'') \in e(.f)\}] \\
t_{[x.f=\texttt{null or new}]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
\hline
t_{[x=y[*]]^n} & = & \lambda e.e[y \mapsto e(y) \cup \{(0,n)\}] \\
 & & \quad [[*] \mapsto e([*]) \cup \{(n,n') | (0,n') \in e(x)\}] \\
 & & \quad [x \mapsto \emptyset] \\
\hline
t_{[x[*]=y]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
 & & \quad [y \mapsto e(y) \cup \{(0,n'') | (n',n'') \in e([*])\}] \\
t_{[x[*]=\texttt{null or new}]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}]
\end{array}
$$

Fig. 3. Environment transformers for path inference

$[x = y.f]^n$. The transformer for this statement performs two tasks. Firstly, it records that the object pointed-to by `y` is being accessed, by adding the transition $0 \xrightarrow{y} n$ to the incoming environment e: $y \mapsto e(y) \cup \{(0,n)\}$. Secondly, it preserves object accesses performed via paths prefixed with the variable `x` by rewriting them to start with `y.f` instead. For example, in `atomic { x = y.f; x.g = 1; }`, to protect the object access `x` in `x.g` at the start of the atomic section, we require locking `y.f`. This is achieved by replacing all transitions of the form $0 \xrightarrow{x} n'$ with the pair of transitions $0 \xrightarrow{y} n$ (already generated above) and $n \xrightarrow{.f} n'$: $.f \mapsto e(.f) \cup \{(n,n') | (0,n') \in e(x)\}$. Finally, we delete `x`'s transitions: $\texttt{x} \mapsto \emptyset$.

$[x.f = y]^n$. This statement accesses the object `x` and modifies its `f` field to point to object `y`. Our transformer records the access by adding it to `x`'s transition set in the incoming environment e: $x \mapsto e(x) \cup \{(0,n)\}$.

With previous statements, we preserve object accesses made below by simply rewriting paths beginning with the lvalue to instead be prefixed with the rvalue. However, this assignment could, in addition to paths starting with `x.f`, also affect paths prefixed with `z.f` for all variables `z` that alias `x`. For example, in `atomic { x.f = y; z.f.g = 1; }`, to protect the access `z.f` in `z.f.g`, there are two possibilities. (i) `x` and `z` are aliases: the atomic section is then the same as `atomic { z.f = y; z.f.g = 1; }`, so we lock `y`. (ii) `x` and `z` are not aliases: the object `z` is not modified by the assignment, therefore the path `z.f` is not affected so we lock `z.f` (and not `y`).

Our analysis uses type information to determine whether two paths may alias each other. In particular, the assignment `x.f = y` affects the path `z.f` if the classes that define the field `f` being accessed in both `x.f` and `z.f` (determined statically in Java) are the same. If they are, we add the path `y`, otherwise we conclude that `z.f` will definitely not be affected and do nothing. Note, even if `x` and `z` may be aliases, the original path `z.f` is not deleted in case they're not.

In general, the affected path may be of the form `v.`$\bar{f}$`.f` where $\bar{f}$ is a sequence of zero or more field lookups that could include `f`. Hence, our transformer adds a transition $0 \xrightarrow{y} n'$ for each $n'' \xrightarrow{f} n'$ transition whereby the field f is the same as that being accessed in `x.f`: $y \mapsto e(y) \cup \{(0, n'') | (n', n'') \in e(.f)\}$. Points-to information would reduce the number of $0 \xrightarrow{y} n''$ transitions but may complicate the composition of transformers.

$[x.f = new]^n$ *and* $[x.f = null]^n$ As type information only tells us if two paths may alias, we can never assert that they definitely must alias. Hence, we cannot assume that accesses of the form `z.f` will be local (`new`) or generate a `NullPointerException` (`null`). We can assume this for paths prefixed with `x.f` as we know `x.f` aliases itself. In this latter case, we would not acquire the lock for `x.f`. To cover both scenarios where we can and can't delete the path, the transformer only adds the access of `x`.

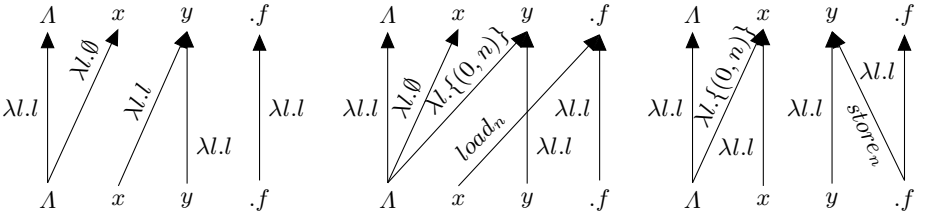
$[x = y[*]]^n$. The transformer for this statement is similar to that for `x = y.f`. We record the access of the array object `y` in the incoming environment e: $y \mapsto e(y) \cup \{(0, n)\}$. We do not distinguish between different array locations representing them all using [*], which can be read as "somewhere in the array." Our transformer preserves object accesses by translating all paths that begin with `x` to start with `y[*]`. We replace each transition $0 \xrightarrow{x} n'$ with the pair $0 \xrightarrow{y} n$ (generated above) and $n \xrightarrow{[*]} n'$: $[*] \mapsto e([*]) \cup \{(n, n') | (0, n') \in e(x)\}$. At run-time, locking `y[*]` involves locking all elements of the array `y`.

$[x[*] = y]^n$. We assume all arrays are aliased, so this assignment could affect all paths that end in `[*]`. When translating, we cannot be sure they refer to the same array location being assigned to. Even in the case of `x[*]`, although we are certain the same array is being modified, the indices may differ. Consequently, our transformer does not delete any paths (like for `x.f = y`) but adds a transition $0 \xrightarrow{y} n'$ for each transition of the form $n'' \xrightarrow{[*]} n'$: $y \mapsto e(y) \cup \{(0, n') | (n'', n') \in e([*])\}$.

3.2 Graph Representation of Transformers

We now present the pointwise graph representations for our transformers. Informally, these graphs describe how the outgoing environment e' is derived from the incoming environment e when passing through a program statement. An edge $d_1 \xrightarrow{f} d_2$ in the graph means that $e'(d_2)$ is obtained from $e(d_1)$, with edge function $f : \mathcal{P}(\mathcal{Q} \times \mathcal{Q}) \to \mathcal{P}(\mathcal{Q} \times \mathcal{Q})$ describing exactly how so. In the simplest case, $f = \lambda l.l$ (the identity function), so $e'(d_2) = e(d_1)$. If $e'(d_2)$ is dependent on multiple $e(d_k)$, the meet of the values (after applying the edge functions) is taken. New values are introduced using the special symbol Λ.

Fig. 4 shows the pointwise representations for $t_{[x=y]^n}$, $t_{[x=y.f]^n}$ and $t_{[x.f=y]^n}$ from Fig. 3 (we assume that $\Sigma = \{x, y, .f\}$). Note: our analysis is backwards. Our analysis has five edge functions:



(a) $t_{[x=y]^n}$ (b) $t_{[x=y.f]^n}$ (c) $t_{[x.f=y]^n}$

Fig. 4. Pointwise representations for Fig. 3 key transformers

1. $\lambda l.\{(n', n'')\}$ for introducing a new automaton transition $n' \xrightarrow{d} n''$. For example, the statement $[x = y.f]^n$ of Fig. 4(b) accesses object y and therefore $e'(y)$ must contain the new pair $(0, n)$. This is represented by the edge $\Lambda \xrightarrow{\lambda l.\{(0,n)\}} y$.
2. $\lambda l.\emptyset$ for killing transitions. For example, in Fig. 4(a), $e'(x) = \emptyset$ corresponds to the edge $\Lambda \xrightarrow{\lambda l.\emptyset} x$.
3. $\lambda l.l$ for copying transitions. The edges $y \xrightarrow{\lambda l.l} y$ and $x \xrightarrow{\lambda l.l} y$ in Fig. 4(a) collectively give that $e'(y) = e(y) \cup e(x)$ (as defined in Fig. 3).
4. $load_n = \lambda l.\{(n, n')|(n'', n') \in l\}$ for preserving object accesses across statements of the form $[x = y.f]^n$ and $[x = y[*]]^n$. In Fig. 4(b), the edge $x \xrightarrow{load_n} .f$ rewrites all paths beginning with `x` to instead begin with `y.f` (the access of `y` is represented with the edge $\Lambda \xrightarrow{\lambda l.\{(0,n)\}} y$). The important thing to note is that n is common in both edges.
5. $store_n = \lambda l.\{(0, n')|(n'', n') \in l\}$ for preserving object accesses across statements of the form $[x.f = y]^n$ and $[x[*] = y]^n$. In Fig. 4(c), $.f \xrightarrow{store_n} y$ adds an access of `y` for every path ending in `.f`. Existing paths ending in `.f` are preserved with the edge $.f \xrightarrow{\lambda l.l} .f$.

3.3 Sparsity

Sparsity is important to keep memory usage down. We keep graphs sparse by not explicitly representing trivial edges of the form $d \xrightarrow{\lambda l.l} d$. These implicit edges should not have to be made explicit, as that would be expensive. However, it turns out that determining whether an implicit edge exists is costly for our analysis. Fig. 5(a) shows an example transformer and Fig. 5(b) is its sparse equivalent. Dashed edges are used for trivial edges. Both x and y have no outgoing edges but while the implicit edge $y \xrightarrow{\lambda l.l} y$ exists, the same is not true for $x \xrightarrow{\lambda l.l} x$. This is because x is killed in the outgoing environment, as represented by the edge $\Lambda \xrightarrow{\lambda l.\emptyset} x$. To determine if an implicit edge $d_i \xrightarrow{\lambda l.l} d_i$ exists, the transitive closure now requires checking whether the edge $\Lambda \xrightarrow{\lambda l.\emptyset} d_i$ exists. This has to be done for all d_i, which will slow down transformer composition tremendously.

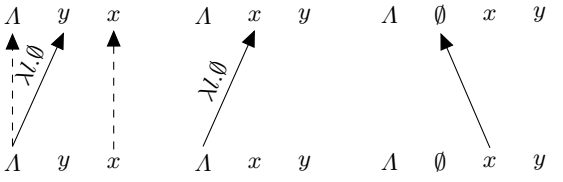


Fig. 5. Determining whether an implicit edge exists is costly

To overcome this problem, we firstly introduce a new special symbol $\emptyset$. Killing the value for symbol d_i in the outgoing environment is then represented with the edge $d_i \to \emptyset$. Secondly, we observe that a large majority of our transformers perform kills, hence we implicitly encode killing within transformer edges. That is, an edge $d_1 \xrightarrow{f} d_2$ now additionally has the meaning $e'(d_1) = \emptyset$. This latter refinement removes the need for kill edges when rewriting paths (e.g. $[x = y]^n$), leading to sparser graphs. The two refinements combined yield the result that an implicit edge $d_i \to d_i$ exists iff d_i has no outgoing edges. Fig. 5(c) shows the refined graph. Symbols Λ, $\emptyset$ and y have no outgoing edges and so each have implicit edges. x has an outgoing edge, therefore has no implicit edge. Fig. 6 shows the refined sparse pointwise representations of Fig. 4. In the case of Fig. 4(c), as we do not kill $.f$ in the outgoing environment, we must add an explicit edge $.f \xrightarrow{\lambda l.l} .f$. However, statements of the form $[x = ...]^n$ are more common, hence the overall effect is that our transformers contain significantly fewer edges.

Transformer Meet. When all edges are explicitly represented, the meet of transformers is graph union. However, when edges are implicitly represented this is not the case and extra care is needed. Fig. 7(a) gives two example transformers whose meet is to be computed. The first transformer preserves all values from the incoming environment to the outgoing environment. The second transformer, however, copies x's value across to y before killing x's value. Hence, the combined transformer should both preserve x's value and also copy it to y. Fig. 7(b) shows the resulting transformer after union, which is not the desired result. This is because graph union is oblivious to the fact that x has an implicit edge in the first transformer. To resolve this, our meet operation makes an implicit edge

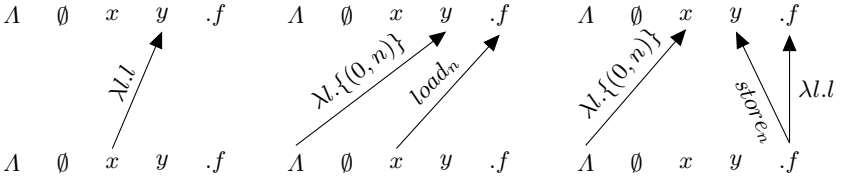


Fig. 6. Refined pointwise representations for Fig. 4

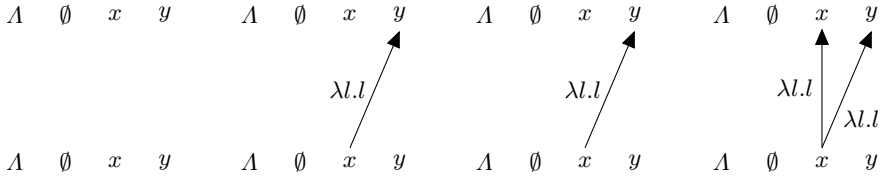


Fig. 7. Computing the meet when implicit edges are present

explicit if at least one other transformer doesn't also have the implicit edge. If none of the transformers have the implicit edge, then it isn't generated in the merged result. The result for this example is shown in Fig. 7(c).

3.4 Native Code, Reflection and Dynamic Loading

All prior lock inference approaches, including our own [14], assume a closed world. Hence, they typically ignore native code, reflection and dynamic loading and assume that the only classes loaded are those that are analysed. We deal with native code/VM calls in this paper like Halpert et al by assuming all effects on the receiver and parameter objects. We do not handle reflection but could approximate the effects of reflective calls by using a tool such as TamiFlex [19], which executes a Java program and produces a reflection trace file. This trace can then be used by the analysis. It would not be possible to acquire locks late in dynamically loaded code, as it may lead to deadlock.

4 Inferring Locks

In this section, we describe how we convert NFAs computed by our analysis to locks. The important challenge here is to balance concurrency and locking overhead. We use instance locks where possible and revert to coarse type locks when we cannot statically determine the set of objects being accessed. We would like to acquire fine-grained locks when we can but still allow the possibility of acquiring type locks when necessary, so we use multi-granularity locking. We also describe how we identify objects that do not need to be locked.

From the summary function computed by our IDE analysis at the start of the atomic section, we extract the NFA describing all objects that may be accessed in the atomic section. We use our previous algorithm [14] to convert the NFA to instance and type locks with the modification that for this paper we use points-to information to determine the possible types of objects involved in cyclic accesses (e.g. linked list traversal), rather than conservatively inferring all types in the class hierarchy rooted at the object's static type.

Our access inference analysis assumes that all object accesses need to be locked. However, there are some objects which do not need to be locked. We identify several classes of such objects: thread-local, instance-local, method-local,

class-local, read-only and dominated objects. We also detect when there is only a single thread executing and avoid taking locks in this case. We now describe each of these.

4.1 Thread-Local Objects (TLA)

An object only needs to be locked if it may be accessed by multiple concurrent threads. We perform a simple analysis to identify objects that are thread-local and do not infer a lock for them. We use Lhotak [20]'s BDD-based thread local analysis implemented in Soot's Paddle framework. This analysis defines all objects reachable from static fields or fields in Runnable classes as being thread shared [34]. It uses Paddle's points-to graph to find these objects.

4.2 Internal Objects (ILA)

Another class of objects we avoid locking are internal objects that exist solely to implement the functionality of another object. An example is the underlying array object used in Java's `ArrayList` implementation. Such internal objects are dominated by their enclosing object `O`, meaning that all accesses to them are performed solely by `O`. This means that to protect accesses to them, when locking is performed outside `O`, it is sufficient to acquire a lock on `O`.

We use a simple and conservative flow-insensitive escape analysis to identify objects that are never accessed outside the instance they are created in. Our escape analysis has two escape modes: *Internal* and *External* (whereby *External* < *Internal*). When an object is created, it is marked as being *Internal* and may become external if:

- It is assigned to a field that is external.
- It is passed as an argument to a method and the receiver object is external or the method is static.
- The object was created in the application's `main()` method or a thread's `run()` method.

A field may become external if:

- It is accessed through an external reference.
- It is assigned an external reference.

Initially, static fields are marked *External*, instance fields are marked *Internal*, non-static method parameters are *Internal* and static method parameters are *External*. We model the return value as assignment to a special return variable `r`, which is initially *Internal* for instance methods and *External* for static methods. For all methods, `this` is always *Internal*. We model array lookups as fields.

Our whole-program analysis finds all reachable methods in the program (including all reachable library methods) and processes them sequentially until a fixed-point is computed. We do not process the call graph in any particular order. We compute per-class and per-method state during fixed-point computation.

Per-class summaries keep track of the escape state of fields, while per-method summaries do so for locals, parameters and the return value. Our analysis can also handle inner classes (as used by iterators) and object handover, such as `A a = new A(new B());` (here the new instance of `B` is being handed-over to the new instanceof `A`).

We use the results of our escape analysis when converting the access NFA to locks by locking the outermost object to protect accesses of internal objects. We can handle multiple levels of internal objects within a single outermost object.

4.3 Single-Threaded Execution

We have found that during the initialisation of an application, many objects are accessed but there is typically only a single-thread executing. Lock acquisitions and releases of our locks can impose significant overheads in this scenario (we do not use thin locks). Thus, we optimise our lock implementation so that locks are treated as no-ops when there is only one thread executing. These object accesses are not thread-local but just that they are only being accessed by a single thread at present. We already remove thread-local locks, as described above.

We detect whether only a single thread is executing or not by incrementing and decrementing a counter when `Thread.start()` and `Thread.join()` are called respectively. If this counter is 0 then we elide the locks otherwise we acquire them as normal. This works because we assume that threads are not spawned by atomic sections.

4.4 Multi-granularity Locks

We use the multi-granularity locking protocol of Gray et al [17] to simultaneously support both type and instance locks. Usually, both coarse- and fine-grained locks cannot protect the same data simultaneously so only one of them would be used. However, multi-granularity locking allows both to be used at the same time for the same data. The multi-granularity locking protocol allows an instance lock to be taken if a coarse-grained type lock protecting the same object hasn't already been acquired and vice-versa. When locking a large number of objects, such as all instances of a type, one can reduce locking overhead by locking the type, whereas in other cases one can lock individual instances to get more concurrency. This orchestration is done at run-time. We implement [21] these locks using Doug Lea's Synchronizer framework [22, 23] for performance.

4.5 Other Optimisations

In addition to removing thread-local and instance-local locks, we perform a number of optimisations to further reduce the locks inferred. This includes analyses for finding: locks that are dominated by other locks (DOM), locks for method-local objects (MLA), locks for objects referred to by static fields that never escape the enclosing class and can therefore be protected by locking the corresponding `Class` object (CLA), objects that are only ever locked in read-mode

and are thus read-only (RO) and finally, types that do not need to be locked in intention mode [17] when their instances are locked (IMP).

Many of these analyses require looking at locks across all atomic sections (e.g. to find out which objects are read-only), therefore we did a final optimisation to ignore atomic sections that are not-reachable from the program's `main` method and will thus never be executed. This greatly improved the results of the previous mentioned analyses.

4.6 Deadlock

Atomic sections must not deadlock as a result of the locks we insert. We acquire locks at the start of the atomic section, allowing us to prevent deadlock at run-time and thus keep per-instance locks, rather than coarsen the locking granularity or impose a static ordering at compile-time and thus potentially hinder concurrency [13, 12]. Given that deadlock rarely occurs, such compile-time approaches to deadlock-avoidance are undesirable.

We avoid deadlock at run-time as follows: when a thread is about to block on a lock l, it first releases all already acquired locks. It then blocks waiting for l to become available after which it starts from scratch to try to acquire all locks (starting from the first lock). This guarantees freedom from deadlock, as it means that locks are not held while waiting, thereby breaking one of the four necessary conditions for deadlock [24].[2] Path expressions must be re-evaluated when re-acquiring locks after waiting because there may have been concurrent updates to the heap. This approach to avoiding deadlock is essentially the same as *retry* used in STM [25]. To improve performance, we use an adaptive locking scheme whereby we first poll l N times before releasing all already acquired locks. Thereafter, we don't block waiting on l but poll until it becomes available before starting the locking stage from scratch as mentioned above.

5 Implementation

We implemented our lock inference approach as a whole-program transformation in Soot (SVN r3588). Here, we give details including optimisations to reduce the memory consumption and running time of our analysis.

5.1 Summary Computation

In this section we describe how we compute per-method summaries. Each method m has a unique entry node N_m and exit node X_m. We store for each CFG node n in m, its local transformer t_n that describes how n transforms environments (see Fig. 3) and an aggregate transformer t_{n,X_m} that summarises the transformation on environments along all execution paths between n and X_m inclusive. The local transformer for a method invocation statement $[x = y.foo(a_1, ..., a_k)]^n$ encapsulates three steps: (i) parameter passing, (ii) invocation of the callee method *foo*

[2] We reduce the likelihood of livelock by using random backoffs.

and (iii) storing the return value to result variable x. Thus, t_n can be expressed as $t_n = t_{n_{params}} \circ t_{n_{invoke}} \circ t_{n_{result}}$. The transformer $t_{n_{invoke}}$ is the summary of the callee *foo*, i.e. T_{foo}. However, due to polymorphism, there may be several possible callees. We therefore take the meet of all such callee summaries.

The summary T_m for a method m is obtained from t_{N_m,X_m} by removing method-local information. Aggregate transformers are computed using a worklist algorithm with two worklists: *intra* and *inter*. Intra consists of nodes whose aggregate transformer needs to be recomputed because the aggregate transformer of at least one intraprocedural successor has changed. Inter contains call nodes n whose invoke transformer $t_{n_{invoke}}$ needs to be updated because the summary of at least one callee has changed. If $t_{n_{invoke}}$ changes as a result, n's aggregate transformer also needs to be recomputed. Per CFG node information is only needed during summary computation after which only the method's summary is kept. Initially, intra contains the exit statement X_m of each method m in the current strongly connected component. Either list is processed exclusively until it becomes empty because interprocedural propagation is expensive and hence it is more efficient to do as much intraprocedural propagation as possible before propagating across method boundaries.

5.2 Reducing Space and Time Requirements

There can be many CFG nodes and transformers can get very large, leading to vast memory usage and slow analysis times. We employ the following techniques to reduce both memory and running time.

Delta Transformers. We observed that after an initial period of propagation, transformers only grow. That is, each time a transformer $(t_n; t_{n,X_m}; T_m)$ is updated, it contains at least the edges it did previously and possibly more. This leads to redundant work because (i) transformer composition is distributive, hence if two edges (one from each transformer) have already been composed before, composing them again will give the same result and (ii) taking the meet is union and unioning old edges gives nothing new. The distributive nature of the analysis thus allows us to process only new edges and then union the results with what has already been computed. We differentiate new edges using a different type of transformer, which we call *delta transformers*. This approach of only propagating additions gives us the biggest speed up and the second best reduction in memory usage.

Summarising CFGs. We implement the technique of Rountev et al [6] that summarises the effects of all execution paths between a pair of CFG nodes n_1 and n_2 in a method m, by combining transformers for statements along these paths. This summary $t_{n_1 \to n_2}$ allows dataflow information to be propagated from n_2 to n_1 (backwards analysis) in one step by composing with it thus reducing propagation and storage. The result of this optimisation is a reduced CFG for m containing three types of nodes: N_m, X_m and recursive calls rc_i together with a set of summary transformers describing effects along execution paths between them.

Parallel Propagation. Another technique we employ to speed up the analysis is to perform propagation in parallel when possible. Our intra worklist contains all CFG nodes that may have to be updated because at least one successor has changed. Although an ordering exists between CFG nodes in the same method, we can exploit the independence between different methods to construct a set of per-method worklists and process the lists in parallel. Our inter worklist contains call nodes that need to be updated when the summary of at least one callee has changed. This involves taking the meet of all callee summaries and then performing parameter-to-argument renaming. There is no dependence between different call nodes in the list so we process them all in parallel.

Efficient Data Structures. Efficient implementations typically use primitives to represent state [22] and manipulate it very quickly using bit-wise operations. We represent transformer edges as 64-bit longs and implement edge composition as a bit-wise operation. However, using primitives with the Java Collections classes leads to boxing/unboxing in/out of their corresponding wrapper classes (e.g. Long), which again is not ideal. We use the Trove library[3], which provides primitive implementations of many data structures such as `HashSet`s and `HashMap`s. We implement transformers as maps, using integers to represent symbols (and sets of longs for their edges).

Worklist Ordering. Ordering the worklist so that successor CFG nodes are given preference over predecessor nodes is an important and well-known optimisation. This makes intuitive sense because dataflow information propagates up the CFG, so if a successor is to be processed again (i.e. it is in the worklist), it may as well be processed before its predecessors in case its value changes once more, thus avoiding unnecessary propagation. We were surprised that this optimisation gave a bigger speed up than CFG summarisation.

6 Evaluation

We now present experimental results for our lock inference approach. We used two experimental machines: (1) *liatris*: a commodity machine consisting of an 8-core 3.4GHz Intel Core i7-2600 CPU, 8GB RAM and running Ubuntu 11.04; and (2) *ax3*: a much larger machine containing 32 8-core 2.67GHz Intel Xeon E7-8837 CPUs totalling 256 cores, 3TB RAM and running SUSE Linux Enterprise Server 11. We use ax3 for analysing `hsqldb` and liatris for analysing all other code and for executing all programs. For running our analysis, we used Oracle's 64-bit JVM and for instrumented runs, we used a modified version of the Jikes RVM. On liatris, we used Oracle JVM version 1.6.0_26-b03 with a minimum/maximum heap size of `4GB` and version 1.7.0_03-b04 with a minimum/maximum heap size of `70GB` on ax3. We did not specify a stack size in either case. For running programs, we used a modified version of the production build of Jikes RVM version 3.1.1+svn (r15989M). Details of the modifications we made are given below.

[3] `http://trove4j.sourceforge.net/`

We begin by giving results for Hello World and use it as a basis for comparing the effect of our different analysis optimisations, as described in Section 5. We demonstrate in Sections 6.4–6.5 that our analysis techniques can scale to large programs by analysing the GNU Classpath library (122KLOC) and the Java database engine hsqldb (150KLOC). Note, for Hello World and GNU Classpath, we used a minimum/maximum heap size of `10GB`.

We evaluate the run-time performance of the benchmarks *sync*, *pcmab*, *bank*, *traffic*, *mtrt* and *hsqldb* instrumented with our locks and compare results with Halpert et al [13], the benchmark's original synchronisation as well as using a single global lock in Section 6.6. Our running times are for all lock optimisations enabled (thread-local, internal object, dominators, etc.). Furthermore, for a fairer comparison, we replace the original `synchronized` blocks and methods with our locks (albeit still maintaining the same locking policy and behaviour as the original `synchronized` blocks).

For fast instance lock retrieval, we modify Jikes by adding an `ilock` field to every object. For fast lookup of type locks, we extend `java.lang.Class` with a `tlock` field. We minimise the additional overhead to object creation by lazily instantiating the lock field. While using a lock table would avoid this, it introduces a lookup overhead which is encountered every time the lock is required. We also extend the `Thread` class for quick access to thread local data (as opposed to using `java.lang.ThreadLocal` that incurs high overhead).

6.1 Soundness of Halpert et al

Halpert et al [13] analyse library call chains upto one level deep and rely on original library synchronisation beyond that. There are many programs where this is sufficient, but it is not sufficient for all problems. Code which has deep library calls fails. Furthermore, if there is no manual synchronisation present then their approach does not guarantee safety of library accesses. For instance, we ran their tool (r3043) on the Hello World program, having removed the existing synchronisation in the library[4] and observed that because they only analyse one level deep they inferred empty read and write sets and when the program executed, print buffers were corrupted causing strings to be printed out multiple times or not at all.[5] Any comparison with their work is a loose one but we do so because it is the closest work to ours.

6.2 Hello World

Although the Hello World program may appear to be a simple one-liner, it requires analysing 1150 methods from the library. Previous work does not fully analyse libraries, hence it is not clear whether existing work can handle this program. Using our own previous work [14], we found it intractable.

[4] See `http://www.doc.ic.ac.uk/~khilan/code/ConcurrentPrintln.java.txt` for the program code

[5] The output of running their tool on Hello World can be found on `http://www.doc.ic.ac.uk/~khilan/code/ConcurrentPrintlnHalpertOutput.txt`

(a) **Analysis (secs)**			(b) **Locks**			
			Instance		*Type*	
Paths	Locks	Total	Read	Write	Read	Write
33	0.6	47	215	54	148	34

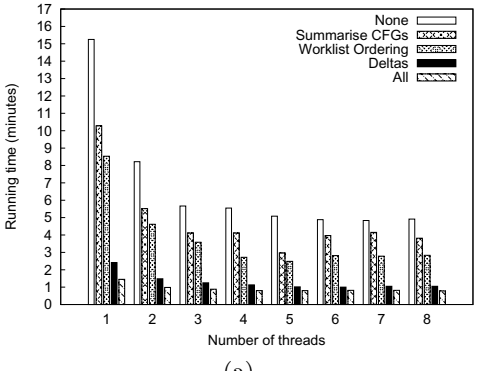
Fig. 8. Analysis results for Hello World

The running times (in seconds) for the path and lock inference analyses are given in Fig. 8(a). The *Total* column gives the time it took to run the whole analysis including Soot-related costs, such as building the call graph and performing the points-to analysis. The times reported are with all analysis optimisations enabled and 8 worker threads. The number of instance read, instance write, type read and type write locks inferred are given in Fig. 8(b). We do not remove any locks. Memory usage peaks at 3.1GB and averages 1.6GB.

6.3 Analysis Optimisations

In this section we evaluate the impact of the analysis optimisations from Section 5.2. We use the Hello World program and compare the effects of delta transformers, CFG summarisation, worklist ordering and parallel propagation on memory usage and running time. All configurations uses the efficient data structures detailed in Section 5.2. Fig. 9 shows our comparison.

The comparison gives a number of interesting insights. Summarising CFGs gives the biggest reduction in memory usage. This is because the number of CFG nodes is significantly reduced and thus so is the amount of analysis state. Secondly, deltas give the best running time performance even if only one thread is used. This is not surprising, because firstly it performs very little redundant work and secondly, as the analysis progresses, the amount of dataflow information propagated reduces thus leading to lesser work over time. Memory usage is also lower because temporary objects are reduced.



(a)

Optimisation	Average MB	Peak MB
None	4923.92	8183.18
Summarise CFGs	2094.68	3470.65
Worklist Ordering	4804.73	8037.14
Deltas	3848.98	6538.27
All	1741.39	3122.84

(b)

Fig. 9. Effect of each optimisation on analysis time (a) and memory usage (b) for Hello World

(a) **Library Info**		(b) **Analysis (secs)**		(c) **Locks**			
				Instance		*Type*	
Package	Methods	Paths	Locks	R	W	R	W
gnu	16882	64.43	9.40	16536	6235	7510	1310
java	13815	46.11	13.67	30065	9940	30007	5354
javax	14088	11.06	6.03	7640	3307	0	0
org	2794	1.31	1.09	1275	401	0	0
sun	28	0.01	0.03	11	4	0	0
Total	47607	127.79	30.22	55527	19887	37517	6655

Fig. 10. Analysis results for GNU Classpath 0.97.2

Also, parallel propagation only gives gains in speed for up to three threads. We think the reason for this is because we process our two worklists in sequence (we do not start processing inter until there is nothing left to do in intra and vice-versa). Consequently, threads that have become free cannot proceed with the other list until all threads have completed. Some methods may require more propagation than others and so this creates a bottleneck.

We were surprised that ordering the worklist so that successors are given preference over their predecessors outperformed the running time when summarising CFGs. This might indicate that unnecessary propagation occurs quite often if worklists are not ordered appropriately.

6.4 GNU Classpath

To evaluate the scalability of our path analysis, we analyse the entire GNU Classpath 0.97.2 library. It consists of 47607 non-private methods and totals about 122KLOC. We analyse each of these non-private methods in turn[6], treating it as an atomic method. We re-use summaries if they have been computed already (during the current analysis run).

We ran our analysis with all analysis optimisations turned on and with 8 worker threads. It took 5 minutes and produced a summary file of size 381MB. Memory usage peaks at 5.1GB and averages 3GB. Fig. 10 gives a per-package breakdown of: (a) number of methods; (b) path inference and lock inference analysis times in seconds and (c) gives the number of each type of lock inferred. Again, we do not remove any locks.

The method which took the longest to analyse was `Logger.getLogger(String)` (30 seconds). Upon inspection, we found that this pulled in the same part of the library as Hello World. Once this set of methods had been analysed, the summaries for methods called by most other methods had already been computed and so did not have to be recomputed. The remaining methods were analysed in a fraction of the time (average of 2ms).

From the locks inferred (Fig. 10(c)), it can be observed that 78% are read locks. This is crucial, as it means that most accesses can proceed in parallel. Furthermore, although nearly 40% of all locks are types, 85% of them are read locks. This again is promising, because it implies that coarse grained locking

[6] Private methods are analysed implicitly with non-private callers.

would not necessarily cripple concurrency (although in the case of Hello World above, we see that the type write locks do).

6.5 HSQLDB

Large real-world programs make extensive use of libraries. We evaluate how well our approach can handle one such program: hsqldb.

This is an SQL relational database engine providing both in-memory and disk-based tables. It is widely used in many open-source as well as commercial products. We use the benchmark version (1.8.0_4) packaged in the Dacapo benchmark suite [26], consisting of an in-memory banking database against which a number of transactions are executed. It comprises a total of 150KLOC and 240 atomic sections (we treat synchronized blocks and methods as atomic sections), as well as making extensive use of GNU Classpath. Fig. 11(a)(i) gives a breakdown of the total number of client and library methods called by atomic sections. Of the 5062 methods called, 58% are in the library.

Our path analysis was able to handle this program after enabling all our analysis optimisations and with a heap size of `70GB`. Memory usage peaked at 64.4GB and averaged 32.4GB. During the ∼7 hours taken to complete the analysis, only 153 seconds (i.e. 2.5 minutes) were spent doing GC. The long analysis time is due to long call chains, large call graph components and consequently vast numbers of transformer edges that are propagated. Unsurprisingly, after the first few atomics had been analysed, the remainder were quicker because a large number of methods were common across atomics. Our lock-removing analyses were able to identify many locks that could be removed, as shown in Fig. 12(a).

6.6 Comparison with Halpert et al

We compare the running times of a selection of benchmark programs transformed using our approach with the closest known existing work of Halpert et al [13] in Fig. 11(a).[7] We choose all benchmarks from their paper that do not use wait/notify (our implementation does not currently support this) and provide analysis and run-time statistics for each. We treat all synchronized blocks and methods as if they are atomics and translate them using our algorithm. For a fair comparison when comparing against manual, global and Halpert et al, we replace synchronized blocks with calls to `lock()` and `unlock()` on our locks instead (we maintain the original locking policy).

An important difference between our approaches is that we analyse library methods in full whereas they only consider accesses upto one level deep in library call chains and rely on original library synchronisation beyond that. Their approach can thus be unsound (see Section 6.1). In Fig. 11(a)(i), we list the number of client and library methods called by atomic sections. Fig. 11(a)(ii)

[7] We do not use their published work [13] but their later improved version [27] that they kindly made available to us. This infers sets of fine-grained locks per atomic whereas in their published version they inferred at most one lock per atomic.

Program	Threads	Atomics		(i) Methods		(ii) Analysis (secs)		(iii) Run (secs)			
		Total	Reachable	Client	Library	Halpert	Ours	Manual	Global	Halpert	Ours
sync	8	2	2	0	0	22	127	69.14	71.22	72.69	56.61
pcmab	50	2	2	2	15	22	127	2.28	3.15	2.28	2.47
bank	8	8	6	6	7	22	127	20.89	19.50	35.69	3.88
traffic	2	24	19	4	63	24	130	2.56	4.22	2.65	4.42
mtrt	2	6	4	67	1324	29	169	0.80	0.82	0.78	0.85
hsqldb	20	240	158	2107	2955	48104	23886	3.25	3.12	3.25	11.39

(a)

Program	Paths (secs)	Locks (secs)	Lock optimisations (secs)						
			TLA	ILA	DOM	CLA	RO	IMP	MLA
sync	0.053	0.0090	0.598	8.441	1.42	3.979	0.0010	0.0	0.0010
pcmab	0.194	0.018	0.603	8.309	1.444	3.855	0.0010	0.0	0.0020
bank	0.151	0.019	0.408	8.177	1.376	3.802	0.0020	0.0010	0.0020
traffic	0.433	0.059	0.569	9.267	1.625	3.861	0.0060	0.0020	0.465
mtrt	33.901	1.902	0.623	9.063	1.741	4.259	0.079	0.03	0.0050
hsqldb	21936.024	1345.859	1.667	28.589	9.597	53.125	1.84	2.724	0.079

(b)

Fig. 11. Analysis and run-time results comparison for a selection of benchmarks from Halpert et al [13, 27]. (a) is an overview of analysis and execution times and (b) gives a breakdown of the time taken for each part of our lock inference analysis. The locks column in (b) gives the time taken to convert NFAs to locks (before optimisations).

compares analysis times (both columns include Soot-related costs). We give a breakdown for the running time of each component in our analysis in Fig. 11(b).

Fig. 12(a) gives a comparison of locks inferred. Fig. 12(a)(i) are the locks inferred by Halpert et al, Fig. 12(a)(ii) the locks we infer and Fig. 12(a)(iii) again shows the locks we infer but this time after applying all our lock optimisations.

Program	(i) Halpert		Ours							
			(ii) No lock opt.				(iii) With all lock opt.			
	Static	*Dynamic*	*Inst.*		*Type*		*Inst.*		*Type*	
			R	W	R	W	R	W	R	W
sync	0	2	1	2	0	0	0	2	0	0
pcmab	0	3	1	5	0	0	0	2	0	0
bank	0	3	0	12	0	0	0	6	0	0
traffic	0	19	33	67	0	0	11	18	0	0
mtrt	1	0	905	268	726	130	0	48	6	66
hsqldb	2	11	32508	24956	26429	10943	1725	4155	9792	8301

(a)

Program	(i) TLA				(ii) ILA				(iii) DOM		(iv) CLA		(v) RO				(vi) IMP		(vii) MLA			
	Inst.		*Type*		*Inst.*		*Type*		*Inst.*		*Inst.*		*Inst.*		*Type*		*Inst.*		*Inst.*		*Type*	
	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W
sync	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	2	0	0	0	0
pcmab	0	1	0	0	1	2	0	0	0	0	0	0	0	0	0	0	1	5	0	0	0	0
bank	0	0	0	0	0	2	0	0	0	3	0	0	0	0	0	0	0	12	0	0	0	0
traffic	0	1	0	0	4	41	0	0	1	0	1	6	31	0	0	0	31	49	0	2	0	0
mtrt	52	5	24	20	92	57	24	60	491	204	119	6	613	0	702	0	560	63	0	0	0	0
hsqldb	464	6045	492	450	2352	3315	1682	2552	19775	13780	4951	487	17948	0	15672	0	15070	2276	0	0	0	0

(b)

Fig. 12. Locks inferred for benchmarks in Fig. 11 by Halpert et al (a)(i) and our approach for both without (a)(ii) and with all our lock optimisations enabled (a)(iii). (b) gives a breakdown of how many locks are removed by each of our lock optimisations.

We give a breakdown of how many locks are removed by each respective lock optimisation in Fig. 12(b). The number for IMP indicates how many instances do not need to intentionally lock their type.

Halpert et al distinguish between two types of lock: (i) *static* locks are known at compile-time and (ii) *dynamic* locks are the same as instance locks. Static locks are not equivalent to our type locks because acquiring a type lock implicitly locks all instances. That is, there is no relationship between static and dynamic locks in their approach. Furthermore, all locks are write locks.

Fig. 11(c) gives execution times. We are noticeably slower for *hsqldb* due to the larger number of locks being acquired. Note, *hsqldb* involves a large number of library methods, which are not analysed by Halpert et al so a direct comparison is not appropriate. At run-time, only 2745 of the 5062 methods (54%) analysed for hsqldb are called. We are looking into using run-time coverage information to reduce the number of locks taken for code paths that are infrequently executed.

7 Related Work

Lock Inference. While software transactional memory remains the popular approach for implementing atomic sections, recent work has also looked at statically inferring locks sufficient for atomic and deadlock-free execution.

In McCloskey et al's Autolocker tool [12], the programmer annotates which locks protect each path expression. Locks are acquired before object accesses and released at the end of the atomic section. Deadlock is prevented statically by ordering path expressions for locks, with the program being rejected if an ordering is not possible. This approach is shown to scale to a 50KLOC web server. Autolocker allows internal objects to be protected by the same lock with a suitable annotation, however our approach differs because we automatically infer these objects. Emmi et al [10] extend upon Autolocker by removing the need for annotations. They have two types of lock: per-instance and per-path expression whereby the latter protects all instances of a path expression and is used when two path expressions p_1 and p_2 may alias each other along some execution path. Lock inference is formulated as an 0-1 ILP problem that aims to minimise the number of locks as well as the number of conflicts between atomic sections. Their approach is shown to scale to 15KLOC. These two approaches do not translate path expressions past assignments and subsequently lock operations cannot be pushed further up without coarsening the locking granularity.

Hicks et al [8] infer *abstract objects* that are each protected by their own lock. This has the advantage that deadlock can be prevented statically, as the number of locks is known at compile-time. However, less concurrency may occur because per-instance locks are not supported. Locks are acquired at the start and released at the end of the atomic section. We base our dominators analysis on theirs but ours differs because we do it for path expressions. Furthermore, we have a working implementation.

Cherem et al [9] also infer path expressions and translate them when pushing through assignments. They also acquire locks at the start of the atomic and

release them at the end. However, there is a major difference: we represent paths as non-deterministic finite state automata, allowing unbounded accesses to be represented precisely, whereas they immediately collapse these accesses to some lock R. The focus of their work is a general theoretical framework for lock inference analyses.

Finally, Halpert et al [13] and Zhang et al [11] take a top-down approach (whereas those previously mentioned and this paper are bottom-up approaches) by instead determining which atomic sections may conflict with each other and then preventing them from proceeding in parallel by allocating to each an appropriate set of locks. These locks may or may not have any relation to the objects being accessed but their purpose is just to prevent conflicting atomics from running concurrently. Bottom-up approaches, like ours, map accesses to locks, which implicitly prevent conflicting atomics from running in parallel. Halpert et al use a May Happen In Parallel [28] analysis to improve the precision of conflict detection and a thread-local/thread-shared analysis to reduce the size of the read/write set of each atomic section. They do not require two-phased locking. However, deadlock is prevented by assigning the same static lock to each atomic section involved in the wait-cycle, thus preventing them from executing in parallel, even if on some/most runs they could. Halpert et al analyse Java programs but only analyse library call chains upto one level deep.

Interprocedural Analysis of Large Programs. The original *callstrings* approach for interprocedural analysis [29] is known to not scale well [30]. Khedker et al [30] propose grouping callstrings into equivalence classes based upon dataflow values and subsequently performing propagation through a method only once per value. We implemented this but were not successful for Hello World. However, this may be due to our implementation of the technique. Regardless of this, the callstrings approach also has the limitation of not allowing pre-computed results for a method to be stored for re-use later, as it does not encode how methods translate dataflow information. Consequently, library methods would have to be re-analysed at each call site.

Recent work [15, 31, 32, 6, 33, 34] has looked at scalable interprocedural analyses using procedure summaries. [15] present an efficient graphical representation for transfer functions and [6] apply this to Java programs that make use of the library. Our work differs from [6] as they present a general framework for whole-program IDE analyses but do not apply it specifically to lock inference . We assume a closed world whereas they consider the possibility of call backs from the library to client code. Our work could be extended to cater for this.

While propagation of delta information is not a new idea, we believe that this is the first time that they have been presented for the IDE framework.

8 Conclusion and Future Work

We believe that this is the first lock inference approach that can analyse precisely programs built with large libraries. Previous lock inference work [14, 11, 13, 12, 8–10] either ignores libraries, requires library implementors to annotate which

locks to take or only consider accesses performed upto one level deep in library call chains [13]. We are able to handle large programs by formulating our previous path inference analysis [14] as an IDE dataflow problem. We have shown that our analysis can scale to 122KLOC when using the pointwise representation of [15, 6] together with a number of optimisations, which in turn we have evaluated. We also analysed the large Java database engine HSQLDB comprising 150KLOC (plus 3000 methods from GNU Classpath).

We have also implemented several analyses to reduce locks inferred, such as for thread-local and internal objects. Our lock inference approach is the first to automatically identify internal objects and elide locks for them. Furthermore, these analyses are conservative but scale to library code and are still able to identify many such objects, which we have shown through the hsqldb benchmark. We detect when only a single thread is executing and elide locks in this case too. This is orthogonal to thread locality because these are locks for shared objects. This reduced our run-times tremendously, as the locking overheads incurred during single-threaded execution were mitigated.

We evaluate the run-time performance of our instrumented programs for a range of benchmarks and compare results with Halpert et al [13]. Halpert et al only analyse library call chains up to one level deep. For benchmarks that involve little library code, we obtain similar performance but for programs that make extensive use of the library, we are slower. However, our approach analyses all library code and is therefore sound, whereas it can be shown that Halpert et al's approach can produce unsound results (see Section 6.1).

In this kind of work there are always ways to improve it. We believe that major areas of a program may rarely be executed and are looking to take advantage of this to reduce the number of locks taken at run-time by delaying the acquisition of locks protecting such cold code regions.

We don't expect our run-times to match those of optimal hand-crafted locks, however for most code they are probably acceptable. More importantly it should be a far simpler task for programmers to annotate blocks of code as atomic than to get them to place locks correctly, and a correctly annotated program will be a deadlock free, race free program.

Acknowledgements. We are grateful to Microsoft for funding this work. We would like to thank Dave Cunningham for the original idea [14] and the belief that reasonable results could be obtained. We are also very appreciative of the detailed discussions we had with Tristan O. R. Allwood and Sophia Drossopoulou and all their helpful advice. We thank Richard Halpert for providing his benchmark programs and scripts. We also thank the entire SLURP research group at Imperial College for interesting discussions about earlier versions of this work. The work would not have been possible without the advice of members of the Soot, Jikes RVM and concurrency-interest mailing lists.

References

1. Lomet, D.B.: Process structuring, synchronization, and recovery using atomic actions. SIGPLAN Not. (1977)
2. Grossman, D.: The transactional memory/garbage collection analogy. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (2007)
3. Cantrill, B., Bonwick, J.: Real-world concurrency. ACM Queue (2008)
4. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal (2005)
5. Larus, J., Rajwar, R.: Transactional Memory (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers (2007)
6. Rountev, A., Sharp, M., Xu, G.: IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 53–68. Springer, Heidelberg (2008)
7. McCabe, T.: A complexity measure. IEEE Trans. on Soft. Eng. (1976)
8. Hicks, M., Foster, J.S., Pratikakis, P.: Lock inference for atomic sections. In: Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT) (2006)
9. Cherem, S., Chilimbi, T.M., Gulwani, S.: Inferring locks for atomic sections. In: PLDI (2008)
10. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL (2007)
11. Zhang, Y., Sreedhar, V.C., Zhu, W., Sarkar, V., Gao, G.R.: Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 141–155. Springer, Heidelberg (2008)
12. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. ACM SIGPLAN Notices (2006)
13. Halpert, R.L., Pickett, C.J.F., Verbrugge, C.: Component-based lock allocation. In: PACT (2007)
14. Cunningham, D., Gudka, K., Eisenbach, S.: Keep Off the Grass: Locking the Right Path for Atomicity. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 276–290. Springer, Heidelberg (2008)
15. Sagiv, Reps, Horwitz: Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. TCS: Theoretical Computer Science 167 (1996)
16. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: CASCON (1999)
17. Gray, J.N., Lorie, R.A., Putzolu, G.R.: Granularity of locks in a shared data base. In: VLDB 1975: Proceedings of the 1st International Conference on Very Large Data Bases (1975)
18. Chan, B., Abdelrahman, T.S.: Run-time support for the automatic parallelization of java programs. J. Supercomput. (2004)
19. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: ICSE 2011: International Conference on Software Engineering (2011)
20. Lhotak, O.: Program Analysis Using Binary Decision Diagrams. PhD thesis
21. Gudka, K., Eisenbach, S.: Fast Multi-Level Locks for Java: A Preliminary Performance Evaluation. In: EC^2 2010: Workshop on Exploiting Concurrency Efficiently and Correctly (2010)

22. Lea, D.: The java.util.concurrent synchronizer framework. Sci. Comput. Program (2005)
23. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley (2006)
24. Magee, J., Kramer, J.: Concurrency: state models & Java programs. Wiley, New York (2006)
25. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2005)
26. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (2006)
27. Halpert, R.L.: Static lock allocation. Master's thesis. McGill University (2008)
28. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statement that happen in parallel. In: SIGSOFT FSE (1998)
29. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications (1981)
30. Khedker, U.P., Karkare, B.: Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 213–228. Springer, Heidelberg (2008)
31. Gulwani, S., Tiwari, A.: Computing Procedure Summaries for Interprocedural Analysis. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007)
32. Rountev, A.: Component-Level Dataflow Analysis. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 82–89. Springer, Heidelberg (2005)
33. Whaley, J., Lam, M.S.: An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 180–195. Springer, Heidelberg (2002)
34. Choi, J., Gupta, M., Serrano, M., Sreedhar, V., Midkiff, S.: Escape analysis for Java. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (1999)

An Analysis of the Mozilla Jetpack Extension Framework

Rezwana Karim[1], Mohan Dhawan[1], Vinod Ganapathy[1], and Chung-chieh Shan[2]

[1] Rutgers University
{rkarim,mdhawan,vinodg}@cs.rutgers.edu
[2] University of Tsukuba, Japan
ccshan@post.harvard.edu

Abstract. The Jetpack framework is Mozilla's newly-introduced extension development technology. Motivated primarily by the need to improve how scriptable extensions (also called addons in Firefox parlance) are developed, the Jetpack framework structures addons as a collection of modules. Modules are isolated from each other, and communicate with other modules via cleanly-defined interfaces. Jetpack also recommends that each module satisfy the principle of least authority (POLA). The overall goal of the Jetpack framework is to ensure that the effects of any vulnerabilities are contained within a module. Its modular structure also facilitates code reuse across addons.

In this paper, we study the extent to which the Jetpack framework achieves its goals. Specifically, we use static analysis to study *capability leaks* in Jetpack modules and addons. We implemented Beacon, a static analysis tool to identify the leaks and used it to analyze 77 core modules from the Jetpack framework and another 359 Jetpack addons. In total, Beacon analyzed over 600 Jetpack modules and detected 12 capability leaks in 4 core modules and another 24 capability leaks in 7 Jetpack addons. Beacon also detected 10 over-privileged core modules. We have shared the details with Mozilla who have acknowledged our findings.

1 Introduction

Several modern browsers support an extensible architecture that allows end-users to enhance and customize the functionality of the browser. Extensions come in a variety of flavors, such as executable plugins to interpret specific MIME formats (*e.g.,* PDF readers, ActiveX, Flash players), browser helper objects, and scriptable addons.

Our focus in this paper is on scriptable extensions for the Mozilla Firefox browser. Such scriptable extensions, also called *addons*, are written in JavaScript, are widely available, and have contributed in large part to the popularity of the Firefox browser and related tools, such as the Thunderbird mail client. As of December 2011, over 7000 addons, supporting a wide variety of functionalities, are available for Firefox via the Mozilla addons page[1]. Popular examples of addons for Firefox include GreaseMonkey [3], which customizes the look and feel of Web pages using user-defined scripts, Firebug [2], which is a JavaScript code development environment, and NoScript [10], which is a security addon that aims to prevent the execution of unauthorized third-party scripts.

[1] http://addons.mozilla.org

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 333–355, 2012.

To support rich functionality, the browser exports an API that JavaScript code in an addon can use to access privileged browser objects and services. On Firefox, this API is called the XPCOM interface (cross-domain component object model) [25], and allows JavaScript code in addon to access a wide variety of services, such as the file system and the network. Access to the XPCOM interface endows JavaScript code in an addon with capabilities that are normally not available to JavaScript code in a Web page. For example, JavaScript code in an addon can freely send `XMLHttpRequest`s to any Web domain, without being constrained by the same-origin policy. The addon can also freely access objects stored on the file system, such as the user's browsing history, cookie store, or any other files accessible by the browser process.

Unfortunately, the privileges endowed by the XPCOM interface can be misused by attacks directed against vulnerable extensions. A recent study of over 2400 Firefox addons [14] found several addons demonstrating insecure programming practices and exploitable vulnerabilities. A successful exploit against vulnerable addons gives the attacker privileges to access the XPCOM interface, via which he can access the rest of the system.

A key problem that has contributed thus far to vulnerabilities and insecure programming of Firefox addons is *the lack of development tools for addon authors*. Addon authors have thus far been required to write their code from scratch, directly accessing the XPCOM interface to perform privileged actions. Such an approach lacks modularity, and provides too much authority to each addon. An exploitable vulnerability anywhere in the addon typically exposes the entire XPCOM interface to the attacker.

To address this problem, Mozilla has recently been developing the *Jetpack framework* [5], officially known as addon SDK [7], a new extension development technology that aims to improve the way addons are developed. It does so using *modularity* and by attempting to enforce *the principle of least authority (POLA)* [26]. A Jetpack addon consists of a number of *modules*. Each module explicitly requests the capabilities that it requires, *e.g.,* access to specific parts of the XPCOM interface, and is isolated from the other modules at the framework level, *i.e.,* its objects are not visible to other modules in the Jetpack addon unless they are explicitly exported by the module. The Jetpack framework therefore aims to contain the effects of vulnerabilities within individual modules by structuring the addon as a set of modules that communicate with each other with clearly defined interfaces, and by ensuring that each module only requests access to the XPCOM interfaces that it needs. The design of the Jetpack addon framework also facilitates code reuse: Jetpack addon authors can contribute the modules used in their addons to the community, following which others can use the modules within their own addons. To bootstrap this process, Mozilla has provided a set of *core modules* that provide a library of features that will be useful for a wide variety of addons.

In this paper, we study the extent to which the Jetpack framework achieves its goals. Specifically, we use static analysis to study *capability leaks* in Jetpack modules and addons. A capability leak happens when a module requests access to a specific XPCOM interface (*i.e.,* a capability), and inadvertently exports a pointer to this interface. Capability leaks allow other modules to access this XPCOM interface (via the exported pointer) without explicitly requesting access to the interface, thereby violating modularity. We also use the same static analysis to study *violations of POLA*, *i.e.,* cases where a module requests access to an XPCOM interface, but never uses it. A vulnerable

module that violates POLA can endow an attacker with more privileges than if the module satisfied POLA.

We applied our analysis to a corpus of over 600 Jetpack modules, which include 77 core modules from Mozilla's Jetpack addon framework and 359 Jetpack addons. Our results show that there are 12 capability leak in 4 core modules and another 24 capability leaks in 7 Jetpack addons. Our analysis also detected 10 core modules violating POLA by requesting privileged resources that they do not utilize. We have shared the details with Mozilla who have acknowledged our findings for the core modules.

2 Background and Motivation

The Jetpack framework [5] focuses on easing the extension development process with an emphasis on modular development, code sharing and security. The framework provides high-level APIs, allowing addon authors the ease of writing extensions using standard Web technologies, like JavaScript and CSS. This is in contrast with traditional extension development, which required developers to be proficient in Mozilla specific technologies like XUL [12] and XPCOM [25].

A Jetpack addon is a hierarchical collection of JavaScript modules, with each module exporting some key functionality. A typical Jetpack addon consists of core modules, user modules and some glue code. Core modules provide low-level functionality and are provided by Mozilla itself. User modules are usually authored by the addon developer or other third-parties who have contributed their code to the community. Glue code ties up all the modules to provide the expected functionality of the addon. On execution, the Jetpack runtime *loads* each component module in a separate sandboxed environment resulting in namespace separation for code within the modules. Inter-module communication is facilitated by special JavaScript constructs, `exports` and `require`, which serve as well-defined entry and exit points for the modules. The `exports` interface enables a module to expose functionality by attaching properties to the `exports` object. The `require` function enables a module to import such exported functionality.

The guidelines by Mozilla advise developers to follow the principle of least authority (POLA) [8] when designing modules. This helps in attenuating the capabilities of modules. The modular architecture of a Jetpack addon coupled with strong isolation between the modules helps to confine the effects of module execution. This is in sharp contrast to the traditional extension development model, where monolithic extensions shared the same namespace and had privileged access to large number of resources via the XPCOM interface. Prior work [15,19] has shown that such extensions are vulnerable to a variety of security threats.

Although not recommended, a Jetpack module may also directly invoke XPCOM interfaces if the desired functionality is not exported by either the core or user modules. However, this is dangerous since interaction with XPCOM interfaces provides access to privileged resources and inexperienced addon authors could inadvertently attach such capabilities to the `exports` interface. Importing such modules would make the requesting module over-privileged and violate POLA.

Figure 1 shows the architecture of a simple Jetpack addon which enables the user to download files from the Web. Each of the dotted boxes in the figure represents a module. Modules such as file, network, preferences represent the core modules and are provided

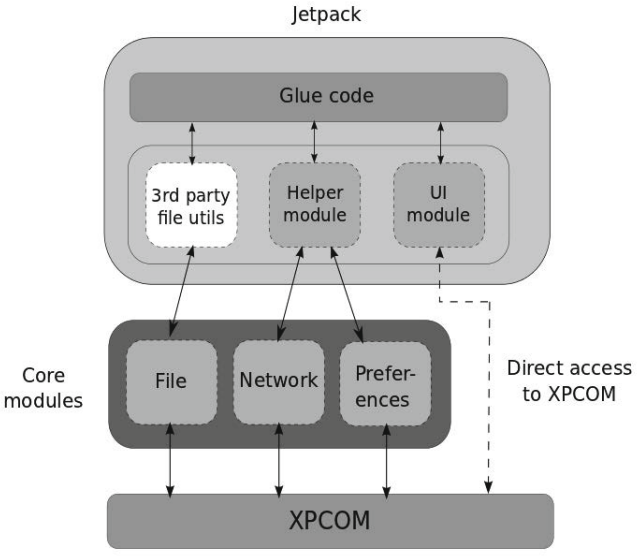


Fig. 1. Structure of a simple Jetpack addon

by Mozilla. The user-level modules include the helper module, the UI module and third-party file utilities. As shown in the figure, the helper module and file utilities build on top of the services exported by the core modules. The UI module directly invokes XPCOM interfaces to support functionality not provided by core modules, such as user alerts or dialog boxes. Although such direct invocations are not recommended (as shown by the dotted line), they are allowed till the Jetpack framework matures and Mozilla develops core modules for all key services.

Capability Leak in a Jetpack Addon

Consider the code snippet as shown in Figure 2 which represents the actual code of the `Preferences` module from 'Customizable Shortcuts' [1], a popular Jetpack addon with over 5000 users. This module exports a method `getBranch` which inadvertently enables access to the browser's entire preference tree. If another module imports the `Preferenecs` module, it would receive additional capabilities to access and modify the user's preferences for all extensions *without explicitly requiring access to the user preferences*; in effect the importing module becomes over-privileged. Although the Jetpack framework recommends adherence to POLA, it does not safeguard against developer mistakes, with the result that unintended capability leaks are frequent.

Let us now examine the code in detail to understand the cause of the capability leak. In line 1, the module requests `chrome` authority to enable it to access ***any*** XPCOM interface. Line 2-11 declare a `Preferences` object with several properties (including `_branches` and `getBranch`) defined on it. On line 12, the module exports the `Preferences` object by attaching it to the `exports` object. Since the entire

```
(1)  const {Cc, Ci} = require("chrome");
(2)  let Preferences = {
(3)    _branches: {},
(4)    _caches: {},
(5)    getBranch: function (name) {
(6)      if (name in this._branches) return this._branches[name];
(7)      let branch = Cc["@mozilla.org/preferences-service;1"]
           .getService(Ci.nsIPrefService).getBranch(name);
(8)      .../* other statements */
(9)      return this._branches[name] = branch;
(10)   }, ... /* other properties */
(11) };
(12) exports.Preferences = Preferences;
```

Fig. 2. Code snippet of a module from a real-world Jetpack addon which leaks the capability to access and modify browser preferences

`Preferences` object is exported, a module which `requires` this module would have access to all its properties, including `getBranch`.

The `getBranch` method utilizes the `chrome` privileges acquired in line 1 to first create an instance of the XPCOM interface `nsIPrefService` and then invoke the `getBranch` method defined on the interface. The `getBranch` method returns an instance of another XPCOM interface `nsIPrefBranch`, which provides a handle to access and modify user preferences. After the assignment in line 7 is complete, `branch` stores an instance of `nsIPrefBranch`. In line 9, the method returns this privileged instance to the caller. Thus, the capability to manipulate the preference tree is leaked through the `exports` interface of the module.

The capability leak from `Preferences` module thus makes an importing module over-privileged, thereby violating POLA. Such a capability leak might even cause inadvertent deletion of user preferences. Ideally, the module should have been designed in a manner to either export access only to its own preference branch, or return primitive values corresponding to the preferences rather than a reference to the branch.

The module also violates another Jetpack addon design principle, which is to utilize capabilities of core modules whenever possible and maintain the hierarchical module structure. The `Preferences` module accesses and returns a reference to the preferences XPCOM interface even though the core modules provide equivalent functionality through the `preferences-service` module, thereby breaking the expected hierarchical structure. The absence of any restriction on developers to use core modules only exacerbates the problem.

Failure to adhere to Jetpack addon guidelines and principles is common in Jetpack modules, in part due to the absence of functionality in core modules and also because of the available choices during module design and implementation. Although adherence to POLA ensures that a module has the minimal set of capabilities required to perform its desired functionality, it is hard to implement in practice due to developer mistakes and refactoring oversights. A capability leak analysis for Jetpack modules would help to identify modules that violate POLA and restrict any security threat only to the concerned module.

Table 1. List of some privileged resources and their access interfaces

Entity	Sensitive attributes and methods
Bookmarks	nsIRDFDataSource
Chrome	Components.classes, Components.interfaces, Components.utils, Components.result
Cookies	nsICookieService, nsICookieManager
Document	window.gBrowser.contentDocument, window.document
Files	nsILocalFile, nsIFile
Passwords	nsIPasswordManager, nsIPasswordManagerInternal.
Preferences	nsIPrefService, nsIPrefBranch
Serivces	nsIIOService, nsIObserverService, nsIPromptService
Streams	nsIInputStream, nsIFileInputStream
Window	nsIWindowMediator, nsIWindowWatcher
XPCOMUtils	nsIModule, generateQI

3 Static Analysis of Jetpack Modules and Addons

In this section we describe a static analysis to detect sources of capability generation in Jetpack modules, flow of capabilities through a module and across the module interface.

The capability leak analysis is an instance of static information flow tracking where taint is modeled as the capability of accessing sensitive sources. A list of the sensitive sources considered in our analysis is given in Table 1. These sources are classified as sensitive as they allow module code to access browser resources and perform privileged operations, such as access to arbitrary DOM elements, read/write access to the cookie and password stores, unrestricted access to the local file system and the network, etc.

In the context of Jetpack modules, an object acquires capabilities if (a) it directly accesses any of the sensitive sources (XPCOM interfaces) or (b) aliases capabilities inherited by the module via an explicit `require` call. In our analysis, an object is marked *privileged* if it directly acquires capabilities, while it is considered *tainted* if it transitively acquires the capabilities.

Both privileged and tainted objects propagate the associated capability through different program paths and can potentially leak it through the module's `exports` interface. Thus, the `exports` interface of each module is an information sink. A module can leak capabilities if it exports:

- direct references to privileged or tainted objects, and/or
- functions that provide references to privileged or tainted objects on invocation or on construction.

To identify capability leaks through module interfaces, we do a flow- and context-insensitive call-graph based static analysis of JavaScript in the module code. Our analysis converts the JavaScript code into the Static Single Assignment (SSA) [18] form and analyzes each SSA instruction. It then processes these facts to perform capability leak analysis. The analysis obtains a degree of flow sensitivity by performing a flow insensitive analysis on an SSA representation of the program.

Our analysis models taint values to flow *upwards* in an object hierarchy i.e. an object is tainted if it itself is tainted or any of its properties are tainted. The key insight is that properties can be accessed given a reference to the parent object but not vice-versa. Thus, for the code snippet in Figure 2, `_branches` in line 9 is tainted because one of its

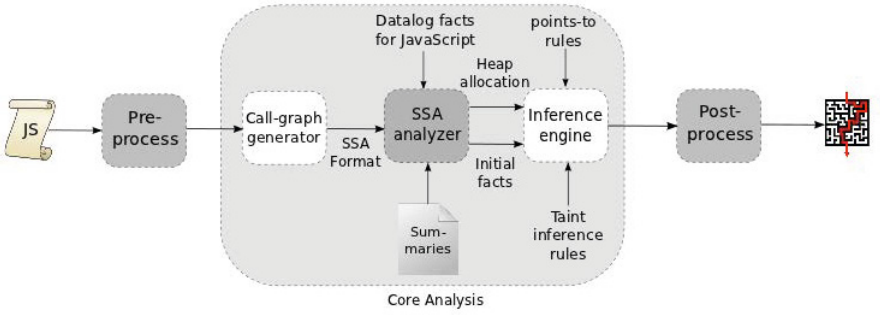


Fig. 3. Overall workflow of our analysis

properties is assigned to `branch`, which is privileged (line 7). Similarly, `Preferences` also gets tainted as one of its children (`_branches`) is tainted. Since there was no capability assignment to `_caches`, it remains untainted.

We have adopted a conservative approach to handle arrays. Since it is statically impossible to precisely determine the index for every array load, store, or access instructions, if any element in the array is tainted then the entire array is marked tainted. Unlike objects, our analysis models all array properties to be tainted if any of the siblings is tainted.

The analysis is inter-procedural. It models functions call sites, arguments and captures the appropriate flow of taint across function invocations. Primitive values are not modeled. Our analysis also does not implement any string analysis. This could affect capability flows arising from string manipulation and dynamic constructs like `eval`. JavaScript containing `eval` is supported, however the code introduced by `eval` is not modeled.

3.1 Stages of the Analysis

Our analysis is based upon Datalog and proceeds in three stages. In the first stage, the analysis pre-processes the addon code to make it amenable to static analysis. The next stage performs the core analysis on the pre-processed code. The core analysis generates Datalog facts that represent capability flow in the Jetpack addon code. The results of core analysis are then processed in the third stage to identify offending flows in the source code of the Jetpack addon. Figure 3 illustrates a schematic diagram of the analysis of a Jetpack addon. The components gray are contributions of this work, while those in white are off-the-shelf tools.

We now describe the various stages of the analysis in detail:

3.1.1 Pre-processing

Our core analysis (as will be described in Section 3.1.2) is based on call-graph construction. The pre-processing stage process the module code to facilitate construction of a complete call-graph for the module.

Since functions are first-class objects in JavaScript and can be properties of other objects, it is possible that such functions are never invoked within the module. Further, if these functions are exported by the module, they could be invoked by the module requesting them. A call-graph generated for such a module would be incomplete since

Table 2. Pre-processed JavaScript constructs and their desugared forms.† We desugar all forms of `let` i.e. statement, expression and definition.

Construct	Desugared to	Code	Desugared Code
Destructuring assignment	property access and assignment	`var {Cc, Ci} = require("chrome");`	`var Cc = require("chrome").Cc;` `var Ci = require("chrome").Ci;`
`let` †	`var`	`let foo = 5;`	`var foo = 5;`
`const`	`var`	`const SIZE = 100`	`var SIZE = 100`
lambda Function	Function	`f(x) x * x`	`f(x) { return x * x; }`

it would not reflect invocations for all the functions. Therefore, we append the module code with additional JavaScript code which would enable the call-graph generator to invoke all functions and generate the complete call-graph for the module. To do so, we consider all functions and properties (including JavaScript getters and setter) reachable from the module's `exports` interface and append appropriate JavaScript statements for their invocation.

We do not append function objects defined in event handling or callback code because the Jetpack runtime freezes the `exports` interface when the module has finished loading. This restricts all event handlers from attaching or modifying `exports` interface. However, the loader does not perform deep freeze of the `exports` object making it possible to modify any property reachable from the interface. Beacon may therefore have false negatives. We plan to extend Beacon to analyze all event handlers.

The pre-processing stage is also required to make the Jetpack addon code amenable for static analysis. To do so, we desugar some of the JavaScript constructs into simpler forms. For example, 'destructuring assignment' is a popular JavaScript construct that mirrors the construction of array and object literals. In essence it only represents syntactic sugar to extract data from arrays or objects. As part of pre-processing, we desugar it and convert it to statements involving simple property access and assignment. For other constructs like `let` and `const`, we change them to `var` statements while keeping the semantics unchanged. Table 2 lists set of the pre-processed constructs along with their desugared forms.

The pre-processing stage also includes code re-writing to simplify statements involving Mozilla specific XPCOM [25] interfaces, which indicate creation or access of privileged resources. To do so, we replace all such XPCOM instances by stubs indicating function calls. For example, the statement in line 7 of Figure 2 is re-written as shown below:

```
let branch = Cc["@mozilla.org/preferences-service;1"]      →     let branch = MozPrefService()
           .getService(Ci.nsIPrefService).getBranch(name);                  .getBranch(name);
```

We also create summaries to indicate capabilities accessible from the stub methods. This summary is fed to the analysis engine to enable it to accurately model the flow of capabilities when handling code that accesses properties on the stub method. For example, the module summary of `MozPrefService` would have one entry for `getBranch` which returns the capability `PrefBranch`.

Table 3. Summary of `preferences` module showing the capability leak

Entity	Type	Capability
`exports`	Object	`prefBranch`
`exports.Preferences`	Object	`prefBranch`
`exports.Preferences._branches`	Object	`prefBranch`
`exports.Preferences.getBranch`	Function	`prefBranch`

3.1.2 Core Analysis

For the purpose of statically analyzing the pre-processed JavaScript code we use an off-the-shelf tool to generate a call-graph in the SSA format. We then generate appropriate Datalog facts corresponding to statements in the JavaScript code and apply inference rules for points-to and capability flow analysis.

Our points-to analysis is inspired by the JavaScript points-to analysis introduced in Gatekeeper [21]. The key distinction is that in our analysis, all program variables carry taint information as well, thereby performing capability flow analysis together with points-to analysis. Similar to prior works [21], we adopt a relatively standard way to represent a program as a database of facts. The set of Datalog relations deployed for the analysis are summarized in Table 4. Each of these relations is of fixed type and arity. The relations specify how points-to and taint information are propagated. We represent heap-allocated objects and functions using the alphabet H, program variables by V, fields by F, call sites by I, integers by Z and capabilities by P.

Unlike prior works [21] which perform whole program analysis, our analysis focuses on modular JavaScript code, such as Jetpack modules. Analysis of individual modules requires that capabilities of each module be appropriately seeded based on which other modules it imports. Since invoking functions from an imported module is akin to using library or foreign functions, we model such functionality as a summary of each module. Thus, a comprehensive analysis of a particular module requires that the summary of each of the imported modules be fed to the analysis engine.

Our analysis focuses primarily on detecting capability flows, thus our summaries only reflect capability leaks possible through the module's `exports` interface, A module's summary typically contains information about the properties of the `exports` interface, their types and taint values reflecting the capabilities associated with the object. Table 3 shows the summary for the code module shown in Figure 2.

Our module summaries simply list the capabilities exported by specific properties exported by a module. In JavaScript, functions can also be exported. However, our summaries are currently not parameterized by the arguments to such functions, which may lead to false negatives in our analysis.

Once summaries for all the imported modules are available, the analysis engine constructs a call graph along with the control-flow graphs for each method in the module to be analyzed. These control-flow graphs consist of several basic blocks which comprise of SSA statements. The analysis engine traverses each of these statements and produces Datalog facts capturing its semantics, as illustrated in Table 5. It also generates heap allocation mappings for the objects and functions, denoted by h_{fresh}. During this phase, several Datalog facts corresponding to the relations shown in Table 4 are generated. The analysis engine then applies the Datalog inference rules presented in Table 6 over

Table 4. Datalog relations used in our static analysis

Relations for points-to analysis		
Heap mapping	ptsTo(V, H)	represents a points-to relation for a variable
	heapPtsTo(H_1, F, H_2)	represents points-to relation for heap objects
	prototypeOf(H_1, H_2)	record object prototype
Object manipulation	assign(V_1, V_2)	represents variable assignments
	store(V_1, F, V_2)	represents field store for an object
	load(V_1, V_2, F)	represents field load from an object
Function manipulation	calls(I, H)	represents call site I invoking method M
	formal(H, Z, V)	represents formal argument of method M
	methodRet(H, V)	represents return value of a method
	actual(I, Z, V)	represents actual parameter of a call site
	callRet(I, V)	represents return value for a call site
Relations for capability flow analysis		
Capability flow	isPrivileged(H, P)	indicates heap object H is privileged with type P
	isTainted(H, P)	indicates heap object H is tainted with type P
	idIsPrivileged(V, P)	indicates variable V is privileged with type P
	idIsTainted(V, P)	indicates variable V is tainted with type P

Table 5. Datalog facts generated for each JavaScript statement

Statement	Example Code	Generated Facts
ASSIGNMENT	$v_1 = v_2$	assign(v_1, v_2)
RETURN	return v	callRet(v)
OBJECT LITERAL	v = {}	ptsTo(v, h_{fresh})
STORE	$v_1.f = v_2$	store(v_1, f, v_2)
LOAD	$v_1 = v_2.f$	load(v_1, v_2, f)
FUNCTION DECLARATION	v = function(v_1, v_2, ..., v_n)	ptsTo(v, h_{fresh}) heapPtsTo(h_{fresh}, prototype, p_{fresh}) for z ∈ 1...n, generate formal(h_{fresh}, z, v_z) methodRet(h_{fresh}, v)
OBJECT CONSTRUCTION	v = new v_0(v_1, v_2, ..., v_n)	ptsTo(v, h_{fresh}) prototypeOf(h_{fresh}, d) :- ptsTo(v_0, h_{method}), heapPtsTo(h_{method}, prototype, d) for z ∈ 1...n, generate actual(i, z, v_z) callRet(i, v)
FUNCTION CALL	v = v_0(v_{this}, v_1, v_2, ..., v_n)	ptsTo(v, h_{fresh}) for z ∈ 1...n, this, generate actual(i, z, v_z) callRet(i, v)

the initial set of facts to keep track of aliases and the flow of capability through the JavaScript code.

Table 6. Datalog inference rules for points-to analysis

Basic rules		
ptsTo(V_1, H)	:-	ptsTo(V_2, H), assign(V_1, V_2)
ptsTo(V_2, H_2)	:-	load(V_2, V_1, F), ptsTo(V_1, H_1), heapPtsTo(H_1, F, H_2)
heapPtsTo(H_1, F, H_2)	:-	store(V_1, F, V_2), ptsTo(V_1, H_1), ptsTo(V_2, H_2)
Call graph		
calls(I, H)	:-	actual(I, 0, V), ptsTo(V, H)
Inter-procedural assignments		
assign(V_1, V_2)	:-	calls(I, H), formal(H, Z, V_1), actual(I, Z, V_2)
assign(V_2, V_1)	:-	calls(I, H), methodRet(H, V_1), callRet(I, V_2)
Prototype handling		
heapPtsTo(H_1, F, H_2)	:-	prototypeOf(H_1, H), heapPtsTo(H, F, H_2).
prototypeOf(O, H)	:-	heapPtsTo(M, prototype, P), heapPtsTo(M, prototype, H), prototypeOf(O, P)
Taint propagation		
isTainted(H_1, P)	:-	heapPtsTo(H_1, F, H_2), isPrivileged(H_2, P)
isTainted(H_1, P)	:-	heapPtsTo(H_1, F, H_2), isTainted(H_2, P)
idIsTainted(V, P)	:-	ptsTo(V, H), isPrivileged(H, P), not(idIsPrivileged(V, P))
idIsTainted(V, P)	:-	ptsTo(V, H), isTainted(H, P)

3.1.3 Post-processing

The combination of initial set of Datalog facts and facts generated after the application of inference rules abstract the behavior of the Jetpack module under analysis. These facts provide information regarding capability flows for the module being analyzed. The post-processing stage links this information back to the source code, identifying possible locations in the source code where capabilities were generated and the properties of the `exports` interface through which they were externalized. This processed information is also utilized for generating a summary for the analyzed module.

3.2 Capability Flow: A Concrete Example

We now demonstrate how the analysis detects capability flows from the `exports` interface of Jetpack addon modules. Figure 4 represents a pre-processed module and the initial set of points-to facts generated by the analysis.

The pre-processed module indicates the use of capabilities within the module by the stub function `MozPrefService`. The ptsTo relations represent object allocations in the heap for each object or function declaration. The analysis engine generates a call-graph with invocation for all methods reachable from the `exports` interface to determine the capabilities flowing out of the module. In the example, the analysis invokes the `exports.Preference.getBranch` method. For brevity, we omit the details of the invocation itself and the associated facts generated for the relevant statements.

	Pre-processed JavaScript statements	Generated Datalog facts
(1)	`var exports = {};`	ptsTo($v_{exports}$, $h_{exports}$)
(2)	`var Preferences = {`	ptsTo($v_{Preferences}$, $h_{Preferences}$)
(3)	`  _branches: {},`	ptsTo($v_{\_branches}$, $h_{\_branches}$) store($v_{Preferences}$, _branches, $v_{\_branches}$).
(4)	`  getBranch: function (name) {`	ptsTo($v_{\_branches}$, $h_{\_branches}$). store($v_{Preferences}$, getBranch, $v_{getBranch}$)
(5)	`    var branch = MozPrefService().getBranch(name);`	ptsTo(v_{branch}, $h_{prefBranch}$).
(6)	`    return this._branches[name] = branch;`	store($v_{\_branches}$, _, v_{branch})
(7)	`  }, ... /* other properties */`	
(8)	`};`	
(9)	`exports.Preferences = Preferences;`	ptsTo($v_{exports}$, $h_{exports}$) store($v_{exports}$, preferences, $v_{Preferences}$)

Fig. 4. Example showing the flow of capabilities through the module's `exports` interface

The analysis detects capability leaks from the module by determining whether `exports` is tainted or not. To do so, it must answer the following Datalog query:

$$\text{idIsTainted}(v_{exports}, X)?$$

where $v_{exports}$ is the SSA representation for the `exports` interface and X is the capability being exported.

Instead of operating on SSA representations, the analysis transforms the above Datalog query to operate on heap allocation representation. Thus, the new query to be resolved is:

$$\text{isTainted}(h_{exports}, X)?$$

where $h_{exports}$ represents heap allocation for $v_{exports}$.

When the analysis invokes the `getBranch` method and analyzes line 5, it reads the summary for `MozPrefService`. This summary lists `getBranch` as method that returns the capability `PrefBranch`. Thus, the analysis engine allocates a heap object ($h_{prefBranch}$) for `nsIPrefBranch` and generates the fact: isPrivileged($h_{prefBranch}$, prefBranch). At line 6, v_{branch} holds the return value of the function `MozPrefService.getBranch(name)`, and thus v_{branch} points to $h_{prefBranch}$. For sake of brevity, we omit the processing of the `return` statement.

On consulting the Datalog inference rules in Table 6 and existing facts, the analysis infers that $h_{prefBranch}$ is stored in the heap allocation object $h_{\_branches}$ thus tainting $h_{\_branches}$. As mentioned earlier in the section, taints propagate upwards in an object hierarchy. Thus the capability `PrefBranch` flows from $h_{\_branches}$ to the heap allocation of the parent object, $h_{Preferences}$ and generates the fact: isTainted($h_{Preferences}$, prefBranch). This in turn generates a similar fact: isTainted($h_{exports}$, prefBranch). Coupled with the fact that $v_{exports}$ points to the heap allocation $h_{exports}$, the analysis resolves X to be `PrefBranch` and determines `PrefBranch` as the capability flowing out of the module through the `exports.Preferences.getBranch` method.

4 Implementation

We realized the analysis described in Section 3 in a tool named Beacon. Beacon is built atop WALA [28], an existing static analysis tool, and uses WALA's capabilities to

Table 7. List of capability leaks observed in the core modules. † indicates multiple reference leaks.

Core module	Capability	Leak mechanism	Essential
tabs/utils †	Active tab, browser window and tab container	Function return	Yes
window-utils †	Browser window	Function return	Yes
xhr	Reference to the `XMLHttpRequest` object	Property of `this` object	No
xpcom	Entire XPCOM utility module	Exported property	No

convert pre-processed JavaScript code into an SSA-based register-transfer intermediate representation (IR) and generate appropriate control-flow graph. Beacon analyzes each IR to generate corresponding Datalog facts, which are processed using the DES Datalog query engine [16]. The core analysis in Beacon was implemented in about 2.8*K* lines of Java code while an additional 700 lines of scripts were required for pre- and post-processing.

5 Results

We evaluated the effectiveness and accuracy of Beacon in detecting capability leaks by analyzing the entire set of 359 Jetpack addons and 77 core modules available to us at the time of writing the paper. In total, Beacon analyzed over 600 modules consisting of over 68*K* lines of JavaScript code. The performance of Beacon's static analysis heavily depends on the size of the analyzed module. On average, Beacon takes a couple of minutes and consumes 200MB per module. For the largest module (`tab-browser.js`/25KB), Beacon took 30mins and 243MB of memory. In Section 5.1 we present results from analysis of the capability leaks in core modules and Jetpack addons. In Section 5.2 we study the nature and usage of capabilities in various Jetpack addons. Lastly, in Section 5.3 we report on the use of Beacon to analyze the privileges associated with Jetpack addons and the core modules to detect over-privileged modules.

Our evaluation methodology involved pre-processing the modules to desugar any incompatible JavaScript constructs and append additional JavaScript code to ensure complete code coverage (see Section 3.1 for details). Each pre-processed module file was individually analyzed by Beacon to generate appropriate Datalog facts that were later processed to extract information about capability leaks. The post-processing also generated a summary for the module that was utilized for analysis of another modules which imported it.

5.1 Capability Leaks

Beacon detected 12 capability leaks in four core modules and another 24 leaks in seven Jetpack addons. Most of the detected leaks were subtle and hard to catch through manual code review. This is reinforced by the fact that Beacon managed to detect 12 capability leaks in production quality code which has undergone numerous code reviews and has a relatively stable code base. For each of the reported leaks, we manually verified the results and observed no false positives. We shared the details of our findings with Mozilla who acknowledged capability leaks in the four core modules. Tables 7 and 8 summarize the findings.

Table 8. Capability leaks in Jetpack addons

Jetpack addon	Capability	Leak mechanism	Essential
Bookmarks Deiconizer	Entire XPCOM services module	Exported property	No
Browser Sign In	window, document	Return from exported function	No
Customizable Shortcut	nsIPrefBranch, nsIAtomService	Property of `this` object	No
	window	Return of function attached to `this`	No
Firefox Share	nsIPrefBranch, window	Property of `this` object	No
	Reference to built-in SQLite database	Property of `this` object	No
	nsIObserverService	Exported property	No
	nsIScriptableInputStream, nsIBinaryInputStream	Return value of exported function	No
	nsISocketTransportService, nsISocketTransport	Property of `this` object	No
	nsIInputStreamPump	Property of `this` object	No
	Instance of the imported `Socket` module	Property of `this` object	No
Most Recent Tab	nsIPrefBranch	Property of `this` object	No
	window	Function return	No
Open Web Apps	nsIPrefBranch, window	Property of `this` object	No
	Reference to built-in SQLite database	Property of `this` object	No
	nsIObserverService	Exported property	No
Recall Monkey	nsIIOService, nsIFaviconService	Property of `this` object	No

Capability Leaks in Core Modules: Beacon discovered two kinds of capability leaks in the core modules. First, capability leaks that occur due to the intended functionality of the module and must therefore be white-listed. Second, capability leaks that occur due to exporting direct references to privileged objects. We list two examples which are representative of the nature of capability leaks in the core modules.

- **window-utils**: The core module `window-utils` as part of its intended functionality exports utility methods to access and track the browser's windows. As mentioned in Section 2, the Jetpack framework executes each module within a sandbox without access to the privileged `window`, `document` or `gBrowser` objects. On analyzing `window-utils`, Beacon reported several capability leaks for methods and properties defined on the `exports` interface that return references to the `window` and `document` objects. Since all of these violations were due to intended functionality as documented in the Jetpack addon SDK [7], we white-listed the offending leaks for the `window-utils` module.
- **xpcom**: The `xpcom` module provides functionality to register a user-defined component with XPCOM and make it available to all XPCOM clients. This module also exposes the `XPCOMUtils` module which offers several utility routines for the components loaded by the JavaScript component loader. Due to the privileged nature of these utility routines, we modeled the `XPCOMUtils` module as a capability source. Our analysis of the `xpcom` module reported a capability leak which we confirmed manually as the reference to the exported `XPCOMUtils` module.
  Exporting a reference to a privileged interface is inconsistent with the philosophy of Jetpack. We believe that instead of the reference to the `XPCOMUtils` module, separate accessor methods that invoke its functionality should be exported by the `xpcom` module. We reported our observation about the `xpcom` module to Mozilla and they agree with our suggestion to wrap the functionality of `XPCOMUtils` with `xpcom` accessors to decrease the surface area for vulnerabilities.

Capability Leaks in Jetpack Addons: Capability leaks discovered by Beacon in the Jetpack addons can be classified into four categories. The first category of leaks occurs due

to export of capabilities through direct references of privileged objects or due to function objects which return capabilities on invocation. The second class of leaks occurs when a module attaches a capability to an exported function's `this` object. The third class of capability leaks occur if the module utilizes the functionality of a core module which itself leaks capabilities, such as `window-utils` or `xpcom`. Lastly, we also observed capability leaks when a Jetpack addon uses third-party modules which themselves leak capabilities. We describe two popular Jetpack addons which demonstrate all four classes of capability leaks.

- **Customizable Shortcuts:** Customizable Shortcuts is a popular Jetpack addon with over 5000 users. It enables users to easily create keyboard shortcuts to customize the Web browser. We analyzed the addon using Beacon and found 3 capability leaks which cover three out of the four classes of leaks. The first leak results from one of the modules exposing a method that on invocation returns reference to the entire preferences tree, instead of the sub-tree specific to the addon. Accessing the entire preferences tree is not recommended since tree modifications on other branches could result in inadvertent loss of user data.

  The second capability leak occurs in a module which exports a wrapper method over the `window-utils` core module. The wrapper invokes functions on `window-utils` which return references to the `window` and `document` objects.

  The last capability leak occurs as a result of the module attaching an instance of the `nsIAtomService` XPCOM interface to the exported function's `this` object. Although, the `nsIAtomService` interface does not provide any security critical functionality, leaking capabilities implicitly through the `this` object is a bad programming practice.

  On manually verifying the leaks, we observed that none of the leaked capabilities was being used by other modules in the Jetpack addon. This suggests that the module author inadvertently exported the capability instead of keeping it local to the module.
- **Firefox share:** Firefox share is a Jetpack addon by Mozilla Labs which allows fast and easy sharing of links from any Web page. This addon has 25 modules with over 5300 lines of JavaScript code. Several of these modules have been reused from another Jetpack addon, Open Web Apps, also by Mozilla Labs.

  Analyzing Firefox share with Beacon, we discovered 10 diverse capability leaks ranging from leaking preference trees, the `window` object, access to a built-in SQLite database to leaking socket services, which would enable a module to leverage benefits equivalent of using raw UDP/TCP sockets. Table 8 enumerates all the observed violations in Firefox share. On manual verification, we observed that in each case the leaked capability was never invoked from any another module. This clearly indicates that the leaks were inadvertent.

  We also found that four of the leaks originated in the code modules that were shared with Open Web Apps. This demonstrates that sharing of over-privileged code modules exacerbates capability leaks.

5.1.1 Accuracy

Beacon detected a total of 36 capability leaks in over 600 modules. For each capability leak, we manually validated the results and observed *no* false positives. However, Beacon could miss capability flows due to a combination of the following reasons:

- **Dynamic features:** Our analysis currently does not handle some of the dynamic and reflective features available in JavaScript. For example, privilege propagation through iterators, generators and reflective constructs like `arguments.callee` are not modeled. Accurate propagation of privileges for such constructs cannot be achieved statically alone and requires dynamic analysis [19,20].
- **Unsupported constructs:** There are a few constructs in JavaScript for which the WALA analysis engine throws exceptions, and thus they are not supported by Beacon. Such constructs include `for..each`, `yield` and `case` statement over a variable. We re-wrote all instances of such constructs (by hand) in the Jetpack modules to make them amenable to analysis. Although hard to quantify, it is possible that the re-written code may miss some capability flows.
- **Unmodeled constructs:** There are some constructs which have not been appropriately modeled yet in our analysis. These include nested `try/catch/throw` sequences, `eval` and `with`. During our experiments, we found ***no*** instance of either `eval` or `with` in any of the modules.

  Also, our analysis currently does not model DOM function calls, like `setAttribute` and property assignments, like `innerHTML`. Such constructs are handled similar to normal JavaScript function calls and property assignments and could affect capability flows.

  Although foreign function calls, like those invoked on imported modules, are modeled, the analysis does not consider the taint value of arguments passed to them. Instead, the analysis determines the taint value of function returns by consulting the module's summary. Ignoring taint values of arguments of foreign functions could also affect the detection of capability flow.
- **Latent bugs:** Lastly, in-spite of exhaustive testing, it is possible that there are latent bugs in Beacon or the automated module summary generation which might affect capability flows.

5.2 Capability Use

The Jetpack framework automatically generates a manifest for each Jetpack addon that provides a dossier about the core modules 'required' by the addon, but provides no information about the XPCOM interfaces invoked by the modules in the addon. As revealed in Section 5.1, a large number of capability leaks originated from the direct use of XPCOM interfaces. In this section, we analyze the Jetpack addons and determine the XPCOM-level capabilities associated with them. A concrete understanding of the capabilities associated with a Jetpack addon is useful to both the end-user and Mozilla itself.

- Addon reviewers at Mozilla can use capability leak analysis to publish fine-grained Jetpack addon manifests that accurately lists all its capabilities. This would be helpful to end-users in making a well-informed choice when installing an addon. For example, if a Jetpack addon invokes the `nsICookieManager` and also has access

Table 9. Top 10 XPCOM interfaces used in Jetpack addons

XPCOM Interface	# Jetpack addons	XPCOM Interface	# Jetpack addons
nsIWindowMediator	18	nsIWindowWatcher	4
nsIIOService	10	nsIFaviconService	4
Services	8	AddonManager	3
nsIPrefService	6	nsILocalFile	3
nsIProperties	5	nsIObserverService	3

to the network, then the end-user can be made aware of the fact that the addon is capable of reading user cookies from all domains and sending them over the network.

– A capability analysis of existing Jetpack addons would help Mozilla in two ways. First, the analysis would identify the set of XPCOM interfaces that are most widely used by developers and for which there do not exist any core modules. This knowledge would help Mozilla in prioritizing the development of core modules. Secondly, the analysis would help the curators at Mozilla to identify addons that use XPCOM interfaces for which a core module already exists. The curator can then suggest the desired modifications to the developer and ensure that all Jetpack addons conform to the hierarchical model where the developer maximizes the use of the built-in core modules for the Jetpack addon functionality.

Table 10. Top 10 core modules used in Jetpack addons

Core module	# Jetpack addons	Core module	# Jetpack addons
self	243	request	101
tabs	160	chrome	94
widget	157	panel	83
page-mod	126	simple-storage	82
context-menu	117	selection	52

To understand the usage pattern of capabilities in Jetpack addons, we modify Beacon to collect two kinds of capability usage characteristics. First, we track all heap object creations that occur when a Jetpack addon invokes an XPCOM interface. Second, we measure the usage of core modules, *i.e.,*the number of core modules imported using a `require` call.

Figure 5 shows the frequency distribution of XPCOM interfaces for the 359 Jetpack addons which directly invoke atleast one XPCOM interface. We observe that 46 of the addons directly invoke XPCOM functionalities, with one Jetpack addon (Firefox share by Mozilla Labs) invoking 14 XPCOM interfaces. Thus over 12% of Jetpack addons directly use XPCOM to include functionality and features not available in the core modules. We believe that as the Jetpack framework becomes popular, this number will increase and along with it the number of modules that leak capabilities.

Tables 9 and 10 list the top 10 XPCOM interfaces and core modules currently in use by Jetpack addons. We observe that 5 of the XPCOM interfaces listed in Table 9, namely `nsIWindowMediator`, `nsIPrefService`, `nsIWindowWatcher`,

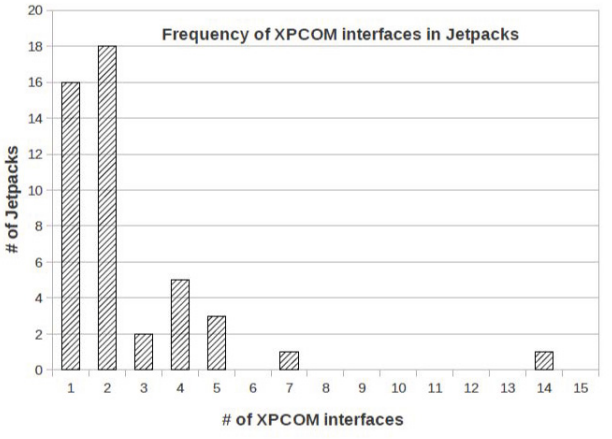


Fig. 5. Frequency of XPCOM interfaces used in Jetpack addons

`nsILocalFile` and `nsIObserverService`, are used by addon authors even though there exist core modules that provide equivalent functionality. For example, the core module `preference-services` provides functionality equivalent to the XPCOM interface `nsIPrefService`. Two of the popular interfaces `nsIIOService` and `Services` provide rich functionality that currently do not have any functionally equivalent core modules. Although a Jetpack addon author can access these capabilities by requesting `chrome` privileges, it increases the privileges associated with the module manifold. The surface area for vulnerabilities in Jetpack addons would greatly reduce if Mozilla could provide core modules for privileged, but frequently used XPCOM interfaces.

Careless handling of multiple capabilities in a module could result in capability leak through the module's `exports` interface. To determine if the modules in Jetpack addons can be split up into better-confined subsets of authority, we used Beacon to detect all modules which accessed more that one XPCOM interface. We grouped the XPCOM interfaces by their functionality and identified modules that used XPCOM interfaces from different categories. If a module uses functionalities from more than one category, then it is a candidate for isolating the authorities used by the module.

We grouped the XPCOM interfaces into 6 categories — namely Application, Browser, DOM, I/O, Security and Miscellaneous — each representing distinct classes of functionalities, All XPCOM interfaces that access application or user preferences, create application threads, etc. are categorized under Application. The category Browser contains interfaces that represent browser neutral functionality like access to timers and console. DOM provides access to the `window` and `document` objects. Services that handle browser permissions and cookies are grouped under Security, while interfaces which require access to the network, file system or storage come under I/O. The remaining interfaces are grouped as Miscellaneous.

We found 26 modules in 19 Jetpack addons, where each module invoked XPCOM interfaces to obtain capabilities of different nature. Table 11 lists the findings. We observe

Table 11. List of Jetpack modules accessing multiple categories of XPCOM interfaces

Jetpack addon	Module name	Categories					
		Application	Browser	DOM	I/O	Security	Misc.
Add-on Builder Helper	main	✓	✓				
	bootstrap	✓			✓		✓
Auto Shutdown NG	countdown	✓		✓	✓		✓
Awesome Screenshot	ui			✓	✓		
Bookmarks Deiconizer	main	✓	✓	✓	✓	✓	✓
Browser Sign In	sessions		✓				
Do Not Fool	localization				✓		✓
Fastest Search	main	✓		✓	✓		✓
Firefox Share	api				✓		✓
	oauthconsumer	✓		✓			
	socket	✓			✓		
	typed-storage				✓		✓
Image2Icon	main		✓	✓			
LepraPanel 2	main		✓	✓	✓	✓	
Memory Meter	main	✓	✓	✓	✓	✓	✓
Open Web Apps	api				✓		✓
	oauthconsumer	✓		✓			
	typed-storage				✓		✓
PriceBlink	main	✓	✓				
Read Later Fast	main		✓	✓			
Recall Monkey	helper			✓	✓		
	main	✓	✓	✓	✓	✓	✓
Snaporama	main	✓	✓	✓	✓	✓	✓
Springpad	main			✓	✓	✓	
Socat	main	✓	✓		✓		✓
Wsad.it Bookmarks	main				✓	✓	

that these modules request a wide variety of authorities, with 4 modules requesting access to all 6 categories. We believe that such modules could be split into better-confined subsets.

5.2.1 Accuracy

We evaluated the accuracy of capability use analysis by comparing the results against the ground truth. By manually analyzing all the modules, we found 53 Jetpack addons which had direct invocations to XPCOM interfaces. Beacon detected 46 addons with XPCOM capabilities. The remaining 7 addons invoked XPCOM interfaces from within event handling code (which Beacon does not model — for reasons stated in 3.1).

5.3 Over-Privileged Modules

The Jetpack addon documentation outlines several guidelines about best practices for developing modules. One of them recommends module authors to follow the principle of least authority (POLA) [8]. To study how the existing core modules conform to this guideline, we analyzed all 77 core modules using Beacon. Our analysis revealed 10 over-privileged core modules.

Table 12 lists the core modules and the nature of the unused privilege. We observe 11 instances of additional privileges which are requested but never utilized in the module code. We also see that 5 of the core modules request critical capabilities like `chrome` and `XPCOM` but never use it. Two modules request `file` and `directory-service` capabilities, which give them privileges to navigate through and read/write to the file system,

Table 12. List of core modules violating POLA

Core module	Privilege	Severity
`file`	Directory service	Moderate
`hidden-frame`	Timer	None
`tab-browser`	Errors	None
`content/content-proxy`	Chrome	Critical
`content/loader`	File	Moderate
`content/worker`	Chrome	Critical
`keyboard/utils`	Chrome	Critical
`clipboard`	Errors	None
`widget`	Chrome	Critical
`windows`	XPCOM, apiUtils	Critical

while the remaining three modules import harmless capabilities which are never used. We contacted Mozilla and notified them about the over-privileged core modules, which they acknowledged as refactoring oversights [6].

5.3.1 Accuracy

To measure the accuracy of false positives in detection of over-privileged modules, we manually validated the Beacon's results for all 77 core modules. Beacon generated a total of 18 warnings for all core modules, out of which 11 were true positives, while the remaining 7 were false positives. On verifying the 7 instances of false positives, we observed that the over-privileged objects were defined in the module's global scope but were used within event handling code. As mentioned in Section 3.1, Beacon does not analyze event handling code, thereby causing false positives.

6 Related Work

Recently, there has been much interest in the analysis of browser extensions for security. To our knowledge, this paper is the first to analyze the Jetpack addon framework.

Sabre [19] and Djeric and Goel [20] both present dynamic information-flow tracking system to detect insecure flow patterns in JavaScript extensions. While the goal of these systems is to detect extensions that can leak sensitive browser data, Beacon instead aims to detect poor software engineering practices in Jetpack modules and addons that can potentially lead to such situations. Moreover, Beacon employs static analysis, which makes it better suited to proactively prevent unwanted information flows in browser extensions.

VEX [13,14] also implements static analysis of JavaScript to study vulnerabilities in extensions. It implements a flow- and context-sensitive analysis that was applied to over 2400 Firefox addons to detect unsafe programming practices. In VEX, vulnerabilities are specified as bad flow patterns; the analysis attempts to verify the absence of these patterns in addons. While VEX was originally applied to traditional Firefox addons, it can also be applied to Jetpack modules to detect bad programming patterns. Beacon's analysis goes further to detect capability leaks that may violate modularity, and violations of POLA, which VEX cannot. Unlike VEX, Beacon employs flow- and context-insensitive analysis of JavaScript. Despite the use of lower-precision analysis, Beacon is able to find real vulnerabilities in Jetpack modules and addons.

IBEX [23] provides tools for extension curators to detect policy violating JavaScript extensions. However, IBEX is a framework for specifying fine-grained access control policies guarding the behavior of monolithic browser extensions, while Beacon performs information-flow for modular JavaScript extensions and is designed to detect modules that violate POLA or leak capabilities across module interface. IBEX also requires extensions to first be written in a dependently-typed language (to make them amenable to verification), following which they are translated to JavaScript. In contrast, Beacon works directly with Jetpack extensions written in JavaScript.

More generally, there has been much recent work on static analysis of JavaScript code executing on Web pages. Beacon borrows and builds upon the techniques introduced in these papers (discussed below), but applies them to the analysis of the Jetpack framework.

The core analysis of Beacon is most similar to that of Gatekeeper [21]. While Gatekeeper was originally applied to study the security of small JavaScript-based widgets, we applied Beacon to study capability leaks in Jetpack addon. Actarus [22] is another static analysis based system that studies insecure flows in JavaScript Web applications. Its set of sources and sinks are thus based on rules targeting specific vulnerabilities. For example, the DOM property `innerHTML` or the method `document.write` is a sink because they facilitate code injection attacks. Beacon in comparison targets Jetpack addon, which have well defined sources (`require` and XPCOM) and sinks (`exports`) for each module. ENCAP [27] is related to Beacon in the domain of identifying capability leaks via static analysis. Like Beacon, ENCAP implements a flow- and context-insensitive static analysis of JavaScript, but Beacon differs in both its implementation and application domain. ENCAP uses static analysis to detect API circumvention, where as Beacon detects capability flows in modular JavaScript code.

Chugh *et al.*present staged information flow [17], an analysis infrastructure for JavaScript code. The goal of their original analysis was to detect insecure flows in JavaScript Web applications. However, they developed a novel phased analysis that would allow new code generated in previous phases to be analyzed. Beacon can possibly use these techniques to analyze dynamic constructs, such as `eval` and `with`.

Although not directly related to the analysis of the Jetpack framework, Google Chrome's extension architecture also encourages a modular design [15]. Its extensions consist of a scriptable part, and a native part, and each extension is required to specify its resource requirements upfront in a manifest. The contents of the manifest are then enforced by the browser, thereby limiting the effect of any exploits against the extension. However, recent works have shown that this model may be insufficient to ensure the security of Chrome extensions [24].

7 Conclusions

In this paper, we described Beacon, a system for capability flow analysis of JavaScript modules. Beacon uses static analysis to detect flow of capabilities through the module's `exports` interface. The techniques used by Beacon are generic, and can detect capability leaks in any modular JavaScript code base, e.g., node.js [9], Harmony modules [4], SproutCore [11]. However, our focus was on browser addons implemented using Jetpack. Beacon cannot directly be applied to non-modular addons.

We implemented Beacon and used it to analyze 77 core modules from Mozilla's Jetpack framework and another 359 Jetpack addons. In total, Beacon analyzed over 600 Jetpack modules and detected 12 capability leaks in 4 core modules and another 24 capability leaks in 7 Jetpack addons. Beacon also detected 10 over-privileged core modules. We have shared the details with Mozilla who have acknowledged our findings for the core modules.

In conclusion, the Jetpack framework attempts to improve how scriptable extensions for the Mozilla Firefox browser are developed. Although it provides guidelines for developing modular addons and recommends POLA, it does not enforce these guidelines. Our evaluation of the Jetpack framework suggests that even heavily-tested core modules may contain capability leaks. The use of a tool such as Beacon during addon development can help prevent such leaks.

The overall security of the Jetpack framework can further be improved by dynamically enforcing permissions requested in extension manifests and by deep freezing the `exports` object. Dynamic enforcement of manifests will ensure that addons are not able to access any resources that they have not explicitly requested. Deep freezing the `exports` object will prevent any capability leak through event handlers. We are investigating other design recommendations in current work.

Acknowledgments. We thank Myk Melez, Brian Warner, David Herman and the whole Jetpack team at Mozilla for helping us in better understanding of the framework. This work was supported in part by NSF grants CNS-0952128 and CNS-0915394.

References

1. Customizable shortcuts, https://addons.mozilla.org/en-US/firefox/addon/customizable-shortcuts/L
2. Firebug: Web development evolved, http://getfirebug.com
3. Greasespot: The weblog about Greasemonkey, http://www.greasespot.net
4. Harmony modules, http://wiki.ecmascript.org/doku.php?id=harmony:modules
5. Jetpack, https://wiki.mozilla.org/Jetpack
6. Jetpack addon refactoring oversights, https://github.com/mozilla/addon-sdk/pull/291
7. Jetpack sdk, https://addons.mozilla.org/en-US/developers/docs/sdk/1.3/
8. Jetpack security model, http://people.mozilla.com/~bwarner/jetpack/components
9. node.js, https://nodejs.org
10. NoScript—JavaScript blocker for a safer Firefox experience, http://noscript.net
11. Sproutcore, http://sproutcore.com/
12. Xul, https://developer.mozilla.org/En/XUL
13. Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M.: Vex: Vetting browser extensions for security vulnerabilities. In: Usenix Security (2010)
14. Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M.: Vetting browser extensions for security vulnerabilities with VEX. CACM 54(9) (September 2011)
15. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: NDSS (2010)

16. Caballero-Roldn, R., Garc-Ruiz, Y., Senz-Prez, F.: Datalog educational system, http://www.fdi.ucm.es/profesor/fernan/des/
17. Chugh, R., Meister, J., Jhala, R., Lerner, S.: Staged information flow in JavaScript. In: ACM SIGPLAN PLDI (2009)
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Kenneth Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 451–490 (1991)
19. Dhawan, M., Ganapathy, V.: Analyzing information flow in javascript based browser extensions. In: ACSAC (2009)
20. Djeric, V., Goel, A.: Securing script-based extensibility inweb browsers. In: Usenix Security (2010)
21. Guarnieri, S., Livshits, B.: GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In: USENIX Security,
22. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R.: Saving the world wide web from vulnerable javascript. In: ISSTA (2011)
23. Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: IEEE S&P (2011)
24. Yan, G., Liu, L., Zhang, X., Chen, S.: Chrome extensions: Threat analysis and countermeasures. In: NDSS (2012)
25. Mozilla Developer Network. Xpcom, http://developer.mozilla.org/en/XPCOM
26. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)
27. Taly, A., Erlingsson, U., Miller, M.S., Mitchell, J.C., Nagra, J.: Automated analysis of security-critical javascript apis. In: IEEE S&P (2011)
28. IBM Watson. Watson libraries for analysis, wala.sourceforge.net/wiki/index.php/Main_Page

Smaller Footprint for Java Collections

Joseph Gil* and Yuval Shimron

Department of Computer Science,
The Technion—Israel Institute of Technology,
Technion City, Haifa 32000, Israel
yogi@cs.technion.ac.il

Abstract. In dealing with the container bloat problem, we identify five memory compaction techniques, which can be used to reduce the footprint of the large number of small objects that make these containers. Using these techniques, we describe two alternative methods for more efficient encoding of the JRE's ubiquitous `HashMap` data structure, and present a mathematical model in which the footprint of this can be analyzed. The *fused hashing* encoding method reduces memory overhead by 20%–45% on a 32-bit environment and 45%–65% on a 64-bit environment. This encoding guarantees these figures as lower bound regardless of the distribution of keys in hash buckets. The more opportunistic *squashed hashing*, achieves expected savings of 25%–70% on a 32-bit environment and 30%–75% on a 64-bit environments, but these savings can degrade and are not guaranteed against bad (and unlikely) distribution of keys to buckets. Both techniques are applicable and give merit to an implementation of `HashSet` which is independent of that of `HashMap`. Benchmarking using the SPECjvm2008, SPECjbb2005 and DaCapo suites does not demonstrate significant major slowdown or speedup. For `TreeMap` we show two encodings which reduce the overhead of tree nodes by 43% & 46% on a 32-bit environment and 55% & 73% on a 64-bit environment. These also give to separating the implementation of `TreeSet` from that of `TreeMap`, which gives rise to footprint reduction of 59% & 54% on a 32-bit environment and 61% & 77% on a 64-bit environment.

1 Introduction

JAVA and the underlying virtual machine provide software engineers with a programming environment that abstracts over many hardware specific technicalities. The *runtime* cost of this abstraction is offset by modern compiler technologies, including just in time compilations [1,5,7,13,15,19,22]. Indeed, there are indications [5,13,19] that JAVA's time performance is approaching that of languages such as C++ and FORTRAN which execute directly on the hardware and are free of potential performance penalties incurred by automatic memory management.

In contrast, memory consumption is not an easy target for automatic optimization. The reason is that there is a direct mapping of programmer-defined

* Corresponding author.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 356–382, 2012.

data structures to runtime layout. An optimizing compiler cannot easily alter programmer defined data structures without disturbing the program semantics. On the other hand, with the ever increasing use of JAVA for implementing servers and other large scale applications, evidence accumulates that the openhanded manner of using memory, which is so easy to resort to in JAVA, leads to memory bloat, with negative impacts on time performance, scalability and usability [17].

The research community response includes methods for diagnosing the overall "memory health" of a program and acting accordingly [18], leak detection algorithms [20], and methods for analyzing [16] profiling [27, 29] and visualizing [23] the heap. (See also a recent survey on the issue of both time and space bloat [28].)

This work is concerned primarily with what might be called "container bloat" (to be distinguished e.g., from temporaries bloat [9]), which is believed to be one of the primary contributors to memory bloat. (Consider, e.g., Mitchell & Sevitsky's example of a large on-line store application consuming $2.87 \cdot 10^9$ bytes, 42% of which are dedicated to class `HashMap` (OOPSLA'07 presentation).)

Previous work on the container bloat problem included methods for detecting suboptimal use of containers [30] or recommendations on better choice of containers based on dynamic profiling [24]. Our line of work is different in that we propose a more compact implementation of containers. Much in the line of work of Kawachiya, Kazunori and Onodera [14] which directly attacks bloat due to the `String` class, our work focuses on space optimization of collection classes, concentrating on `HashMap` and `HashSet`, and to a lesser extent on the `TreeMap` and `TreeSet` classes.

Our work *does not* propose a different and supposedly better algorithmic method for organizing these collections, e.g., by using open-addressing for hashing, prime-sized table, or AVL trees in place of the current implementations. Research taking this direction is rich, but our focus here is on the question of whether *given* data and data structures can be *encoded* more efficiently. After all, whatever method an efficient data structure is organized in, its compaction should lead to an even more frugal use of memory, just as subjecting a super fast algorithm to automatic optimization could improve it further.

To this end, we insist on full compatibility with the existing implementations, including e.g., preserving the order of keys in hash table, and the tree topology of `TreeMap`. Unlike Kawachiya et. al's work, we do not rely on changes to the JVM—the optimization techniques we describe here can be employed by application programmers not only to collections, but to any user defined data structure. Still, the employment of our compaction techniques for aggressive space optimization cannot in general be done without some familiarity with the underlying object model.

We are also motivated by the hope that the techniques we offer could serve optimizing compilers or be employed by other automatic tools for memory optimization (although it is clear that more research is required before all techniques discussed here can thus be exploited).

Table 1.1. Minimal no. of bytes per entry in a set and a map data structures

	32-bit	**64-bit**
Map	8	16
Set	4	8

To appreciate the scale of container overhead, note first that a simple information theoretical consideration sets a minimum of n references for *any* representation of a set of n keys, and $2n$ references for *any* representation of n pairs of key and values.

Table 1.1 summarizes these minimal values for 32-bits and 64-bits memory models, e.g., on a 64-bits memory model no map can be represented in fewer than 16 bytes per entry. Achieving these minimal values is easy if we neglect the time required for retrieval and the necessary provisions for updates to the underlying data structure: A set can be implemented e.g., as a compact sorted array, in which search is logarithmic, while updates consume linear time. The challenge is in an implementation which does not compromise search and update times. Despite recent theoretical results [3] by which one can use, e.g., $n + o(n)$ words for the representation of a dynamic set, while still paying constant time for retrievals and updates, this ideal seems far from being practical: Contemporary implementations of data structures are known to be tolerant of some memory overhead, but, the magnitude of this overhead may be surprising. Consider e.g., `java.util.TreeMap`, an implementation of a red-black balanced binary search tree, which serves as the JRE principal mechanism for the realization of a sorted map datastructure. Memory overheads incurred by this data structure are inferred by examining fields defined for each tree node, realized by the internal class `TreeMap.Entry`.

```
public class TreeMap<K, V> {
  //...
  static final class Entry<K, V>
    implements Map.Entry<K, V> {
      final K key;
      V value;
      Entry<K, V> left = null;
      Entry<K, V> right = null;
      Entry<K, V> parent;
      boolean color = BLACK;
    //...
  }
  //...
}
```

```
public class HashMap<K, V> {
  //...
  static class Entry<K, V>
    implements Map.Entry<K, V> {
      final K key;
      V value;
      Entry<K, V> next;
      final int hash;
    //...
  }
  //...
}
```

Fig. 1.1. Fields defined in `TreeMap.Entry` (a) and in `HashMap.Entry` (b)

Fig. 1.1(a) shows that on top of the object header, each tree nodes stores 5 pointers and a `boolean` whose minimal footprint is only 1 bit, but typically requires at least a full byte (and even an eight bytes word on e.g., the jikes virtual machines). On 32-bits implementation of the JVM which uses 8 bytes per object header and 4 bytes per pointer total memory for a tree node is $8 + 5 \cdot 4 + 1 = 29$ bytes. With the common 4-alignment or 8-alignment requirements (as found in e.g., the HotSpot32 implementation of the JVM), 3 bytes of padding must be added, bringing memory per entry to four times the minimum.

The situation is twice as bad in the implementation of the class `TreeSet` (of package `java.util`), the standard JRE method for realizing a sorted set.

Internally, this class is implemented as proxy to `TreeMap` with a dummy mapped value, bringing memory per entry to eight times the minimum.

A hash-table implementation of the `Map` and `Set` interfaces is provided by the JRE's class `java.util.HashMap` and its proxy class `java.util.HashSet`. Memory per entry of these is determined by two factors. First, as we will explain in greater detail below, each hash table entry consumes $4/p$ bytes (on a 32-bits architecture), where p is the table density parameter ranging typically between 3/8 and 3/4. Secondly, an object of the internal class `HashMap.Entry` (depicted in Fig. 1.1(b)) is associated with each such entry.

On HotSpot32, instances of `HashMap.Entry` occupy 24 bytes, bringing the memory requirements of each `HashMap` entry to the range of 29.33–34.67 bytes in typical ranges of p, i.e., within 10% of memory use with class `TreeMap`.

Table 1.2. Memory overhead per entry of common data structure in central JRE collections

	HotSpot32	HotSpot64
$\mathcal{L}$-`TreeMap`	24	48
$\mathcal{L}$-`TreeSet`	28	56
$\mathcal{L}$-`HashMap`	$16 + 4/p$	$32 + 8/p$
$\mathcal{L}$-`HashSet`	$20 + 4/p$	$40 + 8/p$

Table 1.2 summarizes the overhead, in bytes, for representing the four fundamental JRE collections we discussed. Note that the numbers in the table are of the excess, i.e., bytes beyond what is required to address the key (and value, if it exists) of the data structure.

We describe a tool chest consisting of five memory compaction techniques, which can be used to reduce the footprint of the small `Entry` objects that make the `HashMap` and `TreeMap` containers. These are: null pointer elimination, boolean elimination, object fusion, field pull-up and field consolidation. Techniques can be applied independently, but they become more effective if used wisely together, with attention to the memory model and to issues such as alignments.

Using these techniques, we describe *fused hashing* ($\mathcal{F}$-hash henceforth) and *squashed hashing* ($\mathcal{S}$-hash henceforth): two alternative methods for more efficient encoding of the JRE's implementation of `HashMap` data structure. Fusion and squashing are extended to `TreeMap` as well. Our compaction gives also reason to separating the implementation of `HashSet` from `HashMap` and `TreeSet` from that of `TreeMap`.

We present a mathematical model in which the footprint of these implementations can be analyzed. In this model, we deduce that $\mathcal{F}$-`HashMap` reduces memory overhead by 20%–34% on a 32-bit environment and 48%–54% on a 64-bit environment.

Timing results indicate that no significant improvement or degradation in runtime is noticeable for in three common JVM benchmarks: SPECjvm2008, SPECjbb2005 and DaCapo. Naturally, some specific map operations are slowed down in compare to the simple base implementation due to a more complex and less straightforward implementation.

Table 1.3 summarizes the savings of $\mathcal{F}$-hash, $\mathcal{S}$-hash, $\mathcal{F}$-tree and $\mathcal{S}$-tree. A fully compatible `Map` implementation of the 32-bit and 64-bit versions of $\mathcal{F}$-`HashMap`,

Table 1.3. Memory overhead reduction for a common `Map` instance; implementation of the marked entries is publicly available

		Hash		Tree	
		Fused	*Squashed*	*Fused*	*Squashed*
Map	32-bit	20%–34%†	26%–53%†	43%†	46%†
	64-bit	48%–54%†	32%–56%†	55%†	73%†
Set	32-bit	22%–38%	58%–67%†	59%	54%
	64-bit	50%–62%	64%–73%†	61%	77%

$\mathcal{S}$-`HashMap`, $\mathcal{S}$-`HashSet`, $\mathcal{F}$-`TreeMap` and $\mathcal{S}$-`TreeMap` can be found on the first author's website.These implementations were thoroughly checked correct by the JRE test suite and our own additional testing.

The programming labor in producing the implementations inspired us to present an instrument *virtual entries* that enables transparent inspection of compacted data structures.

A mathematical model was developed for computing the expected savings as a function of the hash table density. This model provides a lower bound on the savings of fused hashing; should the distribution of keys into hash buckets be not as even as the distribution of balls thrown at random into urns, then the actual savings may be greater. For squashed hashing, the model yields an expected value. A non-fully random distribution of keys into buckets may improve, but sometimes worsen the reported savings.

For trees, the balancing algorithms make the reported savings valid for tree with a handful of keys created in any order of key insertions and removals.

A similar analytical approach would not be informative for evaluating time performance. The reason is that the alternative implementations were designed to have the same underlying structure as $\mathcal{L}$-hash: the number of comparisons required to find a key κ is the same in all implementations. Therefore, to evaluate time performance we conducted benchmarking; these were carried out using HotSpot32 (exact settings are described below).

We conjecture that searches in fused and squashed hash tables should be *faster* than the baseline (due to fewer memory dereferencing operations), but that insertions and removals are slower (as they involve repacking of small objects). Initial experiments, not reported here, confirm this conjecture. However, a detailed benchmark timing operations as a function of table density and table size is left for further work, given the intriguing results we, in cooperation with Lenz found [12]. In particular, it was demonstrated that the so called "steady state" which is supposedly reached after sufficient warm-up is not as steady as one might think, with results fluctuating between multiple steady states.

Worse, it was found that the timing of an operation depends on code executed prior to the benchmark. For this reason, the reported benchmarking here focuses on the performance of the compacted data structures as part of a client benchmark application.

Applications using sorted maps and sets are not abundant. Our initial timing of squashed tree data structures indicate that it is as fast, and sometimes

faster than the baseline. However, fused trees are about two times slower; their use should probably be limited to applications operating under strict memory constraints in which either time performance is not a factor, or tree operations are rare.

Outline. The remainder of this article is organized as follows. The five memory compaction techniques we identified are described in Section 2. Sec. 3 then reviews the JRE's implementation class `HashMap`, highlighting the locations in which the optimization can take place based on the statistical properties of distribution of keys into hash table cells. $\mathcal{F}$-hash and $\mathcal{S}$-hash are then described (respectively) in sects. 4 and 5. In Sect. 6 we explain how virtual entries are implemented. Time performance of the compacted hash tables is the subject of Sect. 7. Sect. 8 describes our fused and squashed versions of `TreeMap` and `TreeSet`. Sect. 9 concludes.

2 Compaction Techniques

The space compaction techniques that we identify include the following three:

- *Null pointer elimination.* Say a class C defines an immutable pointer field `p` which happens to be `null` in many of C's instances. Then, this pointer can be eliminated from C by replacing the data member `p` with a non-`final` method `p()` which returns `null`. This method is overridden in a class C_p inheriting from C, to return the value of data member `p` defined in C_p. Objects with `null` values of `p` are instantiated from C; all other objects instantiate C_p.
- *Boolean elimination.* A similar rewriting process can be used to eliminate immutable boolean fields from classes. A boolean field in a class C can be emulated by classes C_t (corresponding to `true` value of the field) and C_f (corresponding to `false`), both inheriting from C.

  In a sense, both null-pointer elimination and boolean elimination move data from an object into its header, which encodes its runtime type. Both however are applicable mostly if class C does not have other subclasses, and even though they might be used more than once in the same class to eliminate several immutable pointers and booleans, repeated application will lead to an exponential blowup in the number of subclasses.

  Mutable fields may also benefit from these techniques if it makes sense to recreate the instances of C should the eliminated field change its value.
- *Object fusion.* Say that a class C defines an ownership [6] pointer in field of type C', then all fields of type C' can be inlined into class C, eliminating the $C \to C'$ pointer. Fusion also eliminates header of the C' object, and the back pointer $C' \to C$ if it exists. It is often useful to combine fusion with null-pointer elimination, moving the fields of C' into C, only if the pointer to the owned object is not `null`.

Before describing the two additional techniques, a brief reminder of JAVA's object model is in place. Unlike C++, all objects in JAVA contain an *object header*, which encodes a pointer to a dynamic dispatch table together with synchronization, garbage collection, and other bits of information. In the HotSpot implementation of the JVM, this header spans 8 bytes on HotSpot32, and 16 bytes on HotSpot64 (but other sizes are possible [4], including an implementation of the JVM in 64-bit environment in which there is no header at all [25]).

The mandatory object header makes fusion very effective. In C++, small objects would have no header (`vptr` in the C++ lingo [10]), and fusion in C++ would merely save the inter-object pointer.

Following the header, we find data fields: `long` and `double` types span 8 bytes each, 4 bytes are used for `int`, 2 bytes for `char` and `short` and 1 byte for types `byte` and `boolean`. References, i.e., non-primitive types take 4 bytes on HotSpot32, and 8 bytes on HotSpot64. Arrays of length m occupy ms bytes, up-aligned to the nearest 8-byte boundary, where s is the size of an array entry. Array headers consume 12 bytes in a 32-bit JVM , 8 for header and 4 for the array `length` field (20 bytes in a 64-bit JVM). Finally, all objects and sub-objects are aligned on an 8-bytes boundary.[1]

Both the header and alignment issues may lead to significant bloat, attributed to what the literature calls *small objects*. Class `Boolean` for example, occupies 16 bytes on Hotspot32 (8 for header, one for the `value` field, and 7 for alignment), even though only one bit is required for representing its content. Applying boolean elimination to `Boolean`, i.e., by making class `Boolean` abstract, while introducing singleton classes `True` and `False` which extend it, would halve the footprint of all `Boolean` objects.

Alignment issues give good reasons for applying the space compaction techniques together. Applying null pointer elimination to class `HashMap.Entry` would not decrease its size (on HotSpot32); one must remove yet another field to reach the minimal saving quantum of 8 bytes per entry.

We propose two additional techniques for dealing with waste due to alignment:

– *Field Pull-up.* Say that a class C' inherits from a class C, and that class C is not fully occupied due to alignment. Then, fields of class C' could be predefined in class C, avoiding alignment waste in C‘, in which the C' subobject is aligned, just as the entire object C. We employ field pull up mostly for smaller fields, typically `byte` sized.

  In a scan of some 20,000 classes of the JRE, we found that the footprint of 13.6% of these could be reduced by 8 bytes by applying greedy field pullup, while 1.1% of the classes would lose 16 bytes. (Take note that field pullup could be done by the JVM as well, in which case, fields of different subclasses could share the same alignment hole of a superclass, and that the problem of optimizing pullup scheme is NP-complete.)

[1] See more detailed description in
`http://kohlerm.blogspot.com/2008/12/how-much-memory-is-used-by-my-java.html` or
`http://www.javamex.com/tutorials/memory/object_memory_usage.shtml`

– *Field Consolidation.* Yet another technique for avoiding waste due to alignment is by consolidation: instead of defining the same field in a large number of objects, one could define an array containing the field. If this is done, the minimal alignment cost of the array is divided among all small objects, and can be neglected.

  Of course, consolidation is only effective if there is a method for finding the array index back from the object whose field was consolidated.

Object fusion was also called *object inlining* in the literature and used for *automatic* optimization of JAVA programs (see Wimmer's Ph.D. thesis [26] for a survey). We suspect that the other techniques enumerated above were employed by programmers without identifying their universality.

3 Hash Tables of the JRE

Other than the implementations designed for concurrent access, we find three principal implementations of hash tables in the JRE: class `IdentityHashMap` uses open-addressing combined with linear probing, i.e., all keys and values are stored in an array of size m, and a new key κ is stored in position $(H(\kappa) + j \bmod m)$ where $H(\kappa)$ is the hash value of κ and j is the smallest integer for which this array position is empty. Class `HashMap` (henceforth called $\mathcal{L}$-`HashMap`) uses *chained hashing* method, by which the i^{th} table position contains a *bucket* of all keys κ for which $H(\kappa) \bmod m = i$. This bucket is modeled as a singly-linked list of nodes of type `HashMap.Entry` (depicted in Fig. 1.1(b)). The hash table itself is then simply an array `table` of type `HashMap.Entry[]`. Finally, class `HashSet` is a wrapper of `HashMap`, delegating all set operations to `map`, an internal `private` field of type `HashMap` that maps all keys in the set to some fixed dummy object value.

It is estimated[2] that class `IdentityHashMap` is 15% to 60% faster than `HashMap`, and occupies around 40% smaller footprint. Yet class `IdentityHashMap` is rarely used[3] since it breaks the `Map` contract in comparing keys by identity rather than the semantic `equals` method. One may conjecture that open addressing would benefit `HashMap` as well. However, Lea's judgment of an experiment he carried in employing the same open addressing for the general purpose `HashMap` was that it is not sufficiently better to commit.

Function H is realized in `HashMap` as `hash(key.hashCode())` where function `hash` is as in Fig. 3.1. This function's purpose is to improve those overridden versions of the `hashCode()` method in which some of the bits returned are less random than others. This correction is necessary since m, the hash table's size, is selected as a power of two, and the computation of $H(\kappa) \bmod m$

```
static int hash(int h) {
 h ^= h >>> 20 ^ h >>> 12;
 return h ^ h >>> 7 ^ h >>> 4;
}
```

Fig. 3.1. Bit spreading function implementation from `HashMap` class

[2] http://www.mail-archive.com/core-libs-dev@openjdk.java.net/msg02147.html msg02147.html

[3] http://khangaonkar.blogspot.com/2010/06/what-java-map-class-should-i-use.html

is carried out as a bit-mask operation. With the absence of this "bit-spreading" function, implementation of `hashCode` in which the lower-bits are not as random as they should be would lead to a greater number of collisions.

Class `HashMap` caches, for each table entry, the value of H on the key stored in that entry. This *Cached Hash Value* (CHV) makes it possible to detect (in most cases) that a searched for key is not equal to the key stored in an entry, *without* calling the potentially expensive `equals` method in function `hash`.

```
public V get(Object κ) {
  if (κ == null)
    return getForNullKey();
  int h = hash(κ.hashCode());
  for (
    Entry<K, V> e = table[h & table.length-1];
      e != null; e = e.next) {
      Object k;
      if (e.h == h
        && ((k = e.K) == κ || κ.equals(k)))
         return e.V;
  }
  return null;
}
```

Fig. 3.2. JAVA code for searching in $\mathcal{L}$-`HashMap`

Function `get` (Fig. 3.2) demonstrates how this CHV field accelerates searching: Before examining the key stored in an entry, the function compares the CHV of the entry with the hash value computed for the searched key. (The listing introduces the abbreviated notation K, V, and h for fields `key`, `value` and `hash`, to be used henceforth.)

A `float` typed parameter known by the name `loadFactor` governs the behavior of the hash table as it becomes more and more occupied. Let n denote the number of entries in the table, and let $p = n/m$. That is what we shall henceforth call *table density*. Then, if p exceeds the `loadFactor`, the table size is doubled, and all elements are rehashed using the CHV. It follows that (after first resize, with the absence of removals), $\texttt{loadFactor}/2 < p \leq \texttt{loadFactor}$. The default value of `loadFactor` is `0.75`, and it is safe to believe [8] that users rarely change this value, in which case, $3/8 < p \leq 3/4$, is an equality we shall call the *typical range of* p, or just the "typical range". The *center of the typical range*, $p = (3/8 + 3/4)/2 = 9/16$ is often used in benchmarking as a point characterizing the entire range.

The memory consumed by the `HashMap` data structure (sans content), can be classified into four kinds:

1. *class-memory.* This includes memory consumed when the class is loaded, but before any instances are created, including `static` data fields, memory used for representing methods' bytecodes, and the reflective `Class` data structure.
2. *instance-memory.* This includes memory consumed regardless of the hash table's size and the number of keys in it, e.g., scalars defined in the class, references to arrays, etc.
3. *arrays-memory.* This includes memory whose size depends solely on the table size.
4. *buckets-memory.* This includes memory whose size depends on the number of keys in the table, and the way these are organized.

Our analysis ignores the first two categories, taking note, for the first category that some of its overheads are subject of other lines of research [21], and for the second, that applications using many tiny maps probably require a conscious optimization effort, which is beyond the scope of this work.

We now compute the *memory use per entry*, i.e., the total memory divided by the number of entries in the table. On HotSpot32 array `table` consumes $4m$ bytes for the array content along with 12 bytes for the array header, which falls in the "instance-memory" category and thus ignored. These $4m$ bytes are divided among the n entries, contributing $4/p$ bytes per entry.

Examining Fig. 1.1(b), we see that each instance of class `Entry` has 8 bytes per header, 3 words for the K, V, and `next` pointers and another word for the CHV, totaling in $8 + 4 \cdot 4 = 24$ bytes per object. The number of bytes per table entry is therefore $24 + 4/p$. For HotSpot64, the header is 8 bytes, the 3 pointers are 8 bytes each and the integer CHV is 4 bytes, which total, thanks to alignment, is $48 + 8/p$ bytes per object. (Comparing these values with Table 1.1 gave the memory overheads of $\mathcal{L}$-`HashMap` and $\mathcal{L}$-`HashSet`, as tabulated in Table 1.2.) Observe that the decision to implement `HashSet` as `HashMap` does not incur any memory toll: eliminating the `value` field gives the same number of bytes per `Entry` object (at least on HotSpot32).

Hashing can be modeled by the famous "balls into urns" statistical model [11], which gives rise to the Poisson distribution: The fraction of hash table buckets with precisely k keys is $\frac{p^k}{k!}e^{-p}$ where $p = n/m$, n being the total number of keys and m the total number of buckets. Fig. 3.3 plots these fractions for $k = 0, \ldots, 4$. The endpoints of the typical range as well as its center are shown as vertical blue lines in the figure. We see that in the typical range a significant portion of the buckets are empty, ranging from 69% (maximal value), to 47% (minimal value). Even when $p = 1$, 37% of the buckets are empty.

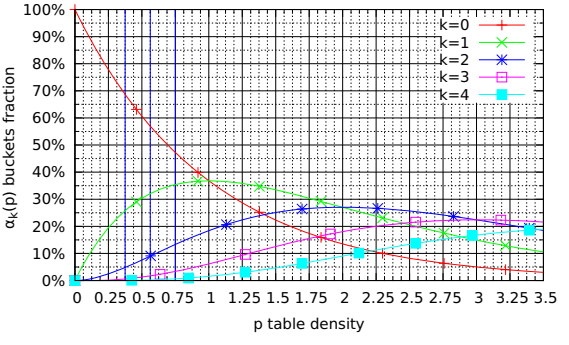


Fig. 3.3. Expected fraction of buckets of size k, $k = 0, \ldots, 4$ vs. table density (and expected fraction of keys falling in buckets of size $k + 1$)

As it turns out, the expected fraction of *keys* which fall into buckets of size $k > 1$, is nothing but the expected fraction of *buckets* of size $k - 1$. We can therefore read the fraction of keys falling into buckets of size k by inspecting the $(k-1)^{th}$ curve in Fig. 3.3. In the range of $p = 3/8$ through $p = 3/4$, we have that the fraction of keys in buckets of size 1 is the greatest, ranging between 69% and 47%. The fraction of keys in buckets of size 2 is between 26% and 35%. At $p = 3/4$, fewer than 15% of the keys fall in buckets of size 3, and, fewer than 4% of the keys are in buckets of size 4.

We identify several specific space optimization opportunities in the $\mathcal{L}$-`HashMap` implementation: First, cells of array `table` which correspond to empty buckets are always `null`. Second, in the list representation of buckets, there is a `null` pointer at the end. A bucket of size k divides this cost among the k keys in it. The greatest cost for key is for singleton buckets, which constitute 47%–69% of all keys in the typical range.

These two opportunities were called "pointer overhead" by Mitchell and Sevitsky [18]. Our fusion and squashing hashing deal with the second overhead (*pointer overhead/entry* in the Mitchell and Sevitsky terminology) but not the first (*pointer overhead/array*): The number of empty buckets is determined solely by p and we see no way of changing this.

Non-null pointers (*collection glue*) are those `next` pointers which are not `null`. These occur in buckets with two or more keys and are dealt with using fusion and squashing. The *small objects overhead* in `HashMap` refers to the fact that *each* `Map.Entry` object has a header, whose size (on HotSpot32) is the same as the essential $\langle K, V\rangle$ pair stored in an entry.

The CHV, the fourth (and last) field of class `HashMap.Entry` (Fig. 1.1(b)), is classified as *primitive-overhead* in the GM taxonomy [18, footnote 4], and can be optimized as well: If $m = 2^\ell$, then the ℓ least significant bits of all keys that are hashed into a bucket i, are precisely the number i. The remaining $32 - \ell$ most significant bits of the CHV are the *only* meaningful bits in the comparison of keys that fall into the same bucket.

We found that storing a `byte` instead of an `int` for the CHV has minimal effect on runtime performance, eliminating 255/256 of failing comparisons. Best results were found for a CHV defined by the coercion `(byte) hash1(key.hashCode())`. where `hash1` is the first stage in computing `hash` (see Fig. 3.4).

```
static int hash1(int h) {
   return h ^ h >>> 20;
}
static int hash2(int h) {
   h ^= h >>> 12;
   return h ^ h >>> 7 ^ h >>> 4;
}
```

Fig. 3.4. Two steps hash code modification function implementation

4 Fused Buckets Hashing

Employing list fusion and pointer-elimination for the representation of a bucket calls for a specialized version of `Entry` for buckets of size k, $k = 1, \ldots, \ell$, for some small integer constant ℓ. The naïve way of doing so is not too effective since in singleton buckets (which form the majority of buckets in the typical range), the specialized entry should include two references (to the key and value) as well as the CHV. With 8-byte alignment, the size of a specialized `Entry` for a singleton bucket is the same as that of the unmodified `Entry`.

Instead, our fused-hashing implementation *consolidates* the CHV of the first key of *all* non-empty buckets into a common array `byte[] chv` of length m, which parallels the main `table` array. If the $\mathtt{i}^{th}$ bucket is empty, then `table[i]` is `null` and `chv[i]` is undefined. Otherwise, `chv[i]` is the CHV of the first key in the $\mathtt{i}^{th}$ bucket, and `table[i]` points to a `Bucket` object that stores the bucket's

contents: for $\mathcal{F}$-`HashMap` this includes all triples $\langle K_j, V_j, h_j \rangle$ that belong in this bucket, with the exception of h_1; for $\mathcal{F}$-`HashSet`, the bucket contents includes all pairs $\langle K_j, h_j \rangle$, with the exception of h_1.

We define four successive specializations of the `abstract` class `Bucket`: `Bucket1` represents singleton buckets and `extends` class `Bucket`; `Bucket2` that `extends` `Bucket1` represents buckets of size 2; `Bucket3` that `extends` `Bucket2`, is dedicated for buckets of size 3; finally, buckets of size 4 or more are represented by class `Bucket4` which `extends` class `Bucket3`. We thus *fuse* buckets of up to four entries into a single object, and employ pointer elimination in buckets of size $k = 1, 2, 3$. In larger buckets, every four consecutive entries are packed into a single object: buckets of size $k > 4$ consist of a list of $\lfloor k/4 \rfloor$ objects of type `Bucket4`. If k is divisible by 4, then the `next` pointer of the last `Bucket4` object of this list is `null`. Otherwise, this `next` field points to a `Bucket`k' object which represents the remaining k' entries in the bucket, where $1 \leq k' \leq 3$ is determined by $k' = k \bmod 4$.

As shown in Table 4.1 in the inheritance chain of `Bucket`, `Bucket1`, ..., `Bucket4` each class adds the fields required for representing buckets of the corresponding length; field pull-up (which depends on the memory model) is employed to avoid wastes incurred by alignment.

Table 4.1. Layout of fused bucket variants in $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` on HotSpot32 and HotSpot64

	HotSpot32				HotSpot64			
	$\mathcal{F}$-`HashMap`		$\mathcal{F}$-`HashSet`		$\mathcal{F}$-`HashMap`		$\mathcal{F}$-`HashSet`	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	object header	8	object header	8	object header	16	object header	16
`Bucket1`	K_1, V_1	16	$K_1, \uparrow h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	16	K_1, V_1	32	$K_1,$	24
`Bucket2`	$K_2, V_2, \uparrow K_3, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	32	$K_2, \uparrow K_3$	24	$K_2, V_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	56	$K_2 \; h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	40
`Bucket3`	$V_3, \uparrow K_4$	40		24	K_3, V_3	72	K_3	48
`Bucket4`	V_4, `next`	48	K_4, `next`	32	$K_4, V_4,$ `next`	96	K_4, `next`	64

For each class, the table shows the introduced fields along with fields pulled-up into it (such fields are prefixed by an up-arrow). The set of fields present in a given class is thus obtained by accumulating the fields introduced in it and all of its ancestors, shown as former table rows. Concentrating on HotSpot32 we see that class `Bucket2` in $\mathcal{F}$-`HashMap` introduces three fields: K_2, V_2 and h_2, but also includes fields K_3 and h_3 which were pulled-up from `Bucket3`, and field h_4 which was pulled-up from `Bucket4`. Class `Bucket2` includes also a h_5 field, which is the CHV of the first key in the subsequent `Bucket` object (or undefined if no such object exists.) The layout of buckets in $\mathcal{F}$-`HashSet`, is similar, except that the absence of value fields increases field pull-up opportunities, leading to greater memory savings.

The "total size" columns in the table suggest that $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` are likely to be more memory efficient than $\mathcal{L}$-hash, e.g., a bucket of size 4 that

requires 96 bytes in $\mathcal{L}$-hash is represented by 48 bytes in $\mathcal{F}$-`HashMap` and only 32 bytes in $\mathcal{F}$-`HashSet`. More importantly, the bulk of the buckets, that is singleton buckets, require only 16 bytes (32 bytes on HotSpot64), as opposed to the 24 bytes (respectively 48 bytes) footprint in the $\mathcal{L}$-hash implementation.

Naturally, objects consume more memory in moving from a 32-bits memory model to a 64-bits model. However, examining the righthand side of Table 4.1 shows that this increase is not always as high as two fold, e.g., a `Bucket1` object doubles up from 16 bytes to 32 bytes in $\mathcal{F}$-`HashMap` but only to 24 bytes in $\mathcal{F}$-`HashSet` (50% increase), and a `Bucket2` object increases from 32 bytes to 56 bytes in $\mathcal{F}$-`HashSet` and from 24 bytes to 40 bytes in $\mathcal{F}$-`HashSet` (both increases are by 67%).

An important property of fusion is that of *non-decreasing compression*, i.e., the number of bytes used per table entry decreases as the bucket size increases. On HotSpot32, overhead per entry in buckets of size 1,2,3,4 is 16, 16, 13.33, 12 bytes in `FHashMap` and 16, 12, 8, 8 in `FHashSet`. In the HotSpot64 model, the respective numbers are 32, 28, 24, 24 bytes in `FHashMap` and 24, 20, 16 ,16 bytes in `FHashSet`. It is easy to check that this property is preserved even in longer buckets.

A search for a given key κ in a fused bucket is carried out by comparing κ with fields $K_1, K_2, \ldots$ in order, and if $\kappa = K_i$ returning V_i, except that K_i is to be accessed if the current `Bucket` object is of type `Bucket`$_i$ or a subtype thereof. It is natural to implement this restriction by overriding function `get` in each of the `Bucket` classes. We found however that, in this trivial inheritance structure, dynamic dispatch is slightly inferior to the direct application of JAVA's `instanceof` operator to determine the bucket's dynamic type. Fig. 4.1 shows some of the details.

```
public V get(Object κ) {
 int h = hash1(κ.hashCode());
 int i = hash2(h) &
  table.length-1;
 Bucket1<K, V> b1 = table[i];
 if (b1==null)
  return null; // Empty bucket
 h = (byte) h;
 Object k;
 if (chv[i]==h && ((k = b1.K1)==κ
  || κ.equals(k)))
   return b1.V1;
 if (!(b1 instanceof Bucket2))
   return null;
 Bucket2<K, V> b2 = (Bucket2)b1;
//...
  if (b4.h4==h && ((k = b3.K4)==κ
   || κ.equals(k)))
    return b3.V4;
  return b4.next==null
         ? null
        : b4.next.get(h, κ, b4.h5);
}
```

Fig. 4.1. JAVA code for searching a key in a fused bucket

In comparing with of $\mathcal{L}$-hash in Fig. 3.2, we see that the first iterations of the loop are unrolled: the search begins with an object b_1 of type `Bucket1`; if κ, the searched key, is different from the K_1 field of b_1, we check whether the current bucket is of type `Bucket2`, in which case, b_1 is down casted into type `Bucket2`, saving the result in b_2, proceeding to examining field K_2, etc. As before the CHV fields $(h_1, h_2, \ldots)$ are used to accelerate the search, and as before an identity comparison precedes the call to the potentially slower `equals` method.

It is generally believed that search operations, and in particular, successful search, are the most frequent operations on collections.[4] This is the reason we did not try to similarly optimize insertions (method `put`) and removals (method `remove`); these were implemented by dynamic dispatch into appropriate methods in the `Bucket` hierarchy.

We turn to space analysis. For that we employ an analytical model to compute the expected number of bytes per table entry. This kind of analysis is justified by the fact that empirical results generally agree with the theoretical model of buckets' distribution: This is true for the initial implementation of function `hashCode` in class `Object`, which on HotSpot is by a pseudo-random number generator. Class designers, particularly of common library classes such as `String`, usually make serious effort to make `hashCode` as randomizing as possible. It is also known [2] that the design of a particularly bad set of distinct hash values is difficult. Finally, the bit-spreading preconditioning of function `hash` (Fig. 3.1), compensates for suboptimal overriding implementations of `hashCode`.

Note that a distribution of keys among buckets which is not random means that buckets tend to be fewer and larger than that predicted by the Poisson distribution. The "non-decreasing compression" property of $\mathcal{F}$-hash guarantees that the analytical model is a lower-bound on the memory savings which can only be larger in practice. For example, in the extreme case in which function `hashCode()` always returns 0, all entries will fall in the first bucket, each map entry will consume only 12 bytes. (The same lower bound could not be stated for $\mathcal{S}$-`HashMap` in which the non-decreasing compression property does not hold.)

A detailed analysis of the space overhead reduction is given in the appendix, for $\mathcal{F}$-`HashMap` on HotSpot32, the expected number of bytes per table entry is

$$12 + \left(9 - 4\cos p \cdot e^{-p}\right)/p.$$

We see that throughout the entire "typical" range, list fusion improves memory use for the hash data structure, reducing it by about a third at $p = 3/4$, and that the improvement increases with p. As it turns out, fusion improves upon the baseline representation throughout the entire typical range as reported above in Table 1.3. Further, this improvement increases monotonically with p; for slightly larger values of p (e.g., in load factor $p \approx 1.5$ in which buckets are still very small) both $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` are close to their asymptotic utility, requiring just a little over 12 bytes of overhead ($\mathcal{F}$-`HashMap`) and just little over 8 bytes of overhead ($\mathcal{F}$-`HashSet`), thus reaching a two (three) fold improvement over the 24 bytes of overhead in $\mathcal{L}$-hash.

The same behavior is exhibited by the 64 bit memory model: significant improvement in the typical range (which surpasses that seen in the 32 bit model), and reaching the same two- or three- fold improvement for larger values of p.

[4] See for example discussion in `http://mail.openjdk.java.net/pipermail/core-libs-dev/2009-June/001807.html`

5 Squashed Buckets Hashing

Squashed buckets hashing further reduces the footprint of fused. The enabling observation is that a singleton bucket does not need to be represented by an object. Consider a cell in array `table` data structure, whose associated bucket is a singleton. Instead of storing in the cell a reference to a singleton bucket object, squashing means that the cell references the *key* residing at this bucket, while the *value* is consolidated into a table-global array. Thus, a hash map consists of three arrays of length m: `keys` and `values` of type `Object`, and, as before, array `chv` of `bytes` storing the CHV of the first key in buckets.

If the $\texttt{i}^{th}$ bucket is empty, then `keys[i]` and `values[i]` are `null`. If it is a singleton, `keys[i]` is the key stored in this bucket, `values[i]` is the associated value, while `chv[i]` is the CHV of this key. Otherwise, bucket `i` has k entries for some $k \geq 2$. In this case, cell `keys[i]` references a `Bucket` object, which must store all triples $\langle K_j, V_j, h_j \rangle$, for $j = 1, \ldots, k$, that fall in this bucket, except that V_1 is stored in `values[i]`, and h_1 is stored in `chv[i]`.[5]

As before, the `Bucket` object is represented using list fusion: class `Bucket2` (which `extends` the `abstract` class `Bucket`), stores the fused triples list when the bucket is of size 2; class `Bucket3`, which `extends` class `Bucket2`, stores the fused triples list when the bucket is of size 3, etc. Class `Bucket6`, designed for the rare case in which a bucket has 6 keys or more, stores the first *five* triples and a reference to a linked list in which the remaining triples reside. For simplicity, we use standard `Entry` objects to represent this list. (A little more memory could be claimed by using, as we did for $\mathcal{F}$-hash, one of `Bucket2`, ..., `Bucket6` for representing the bucket's tail; this extra saving is minute.)

A squashed `HashSet` is similar to a squashed `HashMap`, except for the obvious necessary changes: There is no `values` array, and a `Bucket` object for a k-sized bucket stores pairs $\langle K_i, h_i \rangle$, for $i = 1, \ldots, k$ (h_1 is still stored in `chv[i]`).

Table 5.1 lists the introduced -and pulled-up- fields in classes `Bucket`, `Bucket2`, ..., `Bucket6` in $\mathcal{S}$-`HashSet` and $\mathcal{S}$-`HashMap` on HotSpot32 and HotSpot64.

Of the twenty concrete classes described in the table, only four consume unused space: `Bucket6` of $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` and `Bucket2` of $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet`. The global waste due to the first two classes is meager since buckets with six keys or more are rare, and the waste is divided among all keys in the bucket. However, the effectiveness of squashed hashing on HotSpot64 in somewhat limited by the waste in buckets of size 2.

Observe that since singleton buckets do not occupy any memory, the non-decreasing compression property of fused hashing is not preserved. In other words, unlike fusion, a key distribution in which all keys fall in distinct buckets is the most memory efficient among all other distributions, and when more keys are added to a bucket it does not necessarily become more efficient in reducing the memory overhead per key.

[5] Squashed hashing does not allow keys whose type inherits from class `Bucket`; this is rarely a limitation as this class is normally defined as an inner `private` class of `HashMap`.

Table 5.1. Layout of squashed bucket variants in $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` on HotSpot32 and HotSpot64

	HotSpot32				HotSpot64			
	$\mathcal{S}$-`HashMap`		$\mathcal{S}$-`HashSet`		$\mathcal{S}$-`HashMap`		$\mathcal{S}$-`HashSet`	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	object header	8	object header	8	—	16	—	16
`Bucket2`	$K_1, K_2, V_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	24	$K_1, K_2, \uparrow K_3, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	24	$K_1, K_2, V_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	48	$K_1, K_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	40
`Bucket3`	K_3, V_3	32		24	K_3, V_3	64	K_3	48
`Bucket4`	K_4, V_4	40	$K_4, \uparrow K_5$	32	K_4, V_4	80	K_4	56
`Bucket5`	K_5, V_5	48		32	K_5, V_5	96	K_5	64
`Bucket6`	`next`	56	`next`	40	`next`	104	`next`	72

A search for a given key κ in a squashed bucket is carried out by comparing κ with fields $K_1, K_2, \ldots$ in order, and if $\kappa = K_i$ returning V_i. Unlike fused buckets, this search cannot be implemented solely by dynamic dispatch since no `Bucket` object exists for singleton buckets. Our implementation (Fig. 5.1) deals with singleton buckets by overriding the `equals` method of `Bucket`; the alternative of using `instanceof` and then checking for equality, is possible, but unlikely to be as efficient.

```
public V get(Object κ) {
 int h = hash1(κ.hashCode());
 int i = hash2(h) &
  keys.length - 1;
 h = (byte) h;
 Object k = keys[i];
 if (k==null) return null;
 if (chv[i]==h && (k==κ
   || k.equals(κ)))
    return values[i];
 if (!(k instanceof Bucket2))
  return null;
 Bucket2<K, V> b = (Bucket2) k;
 if (b.h2==h &&
  ((k = b.K2)==κ || κ.equals(k)))
   return b.V2;
 if (!(b instanceof Bucket3))
  return null;
 Bucket3<K, V> b = (Bucket3) b;
 //...
}

class Bucket2<K, V> extends Bucket<K, V> {
 K K1, K2;
 V V2
 byte h2, h3, h4, h5;
 //...
 @Override final boolean equals(Object κ) {
   return K1==κ || K1.equals(κ);
 }
 //...
}
```

Fig. 5.1. JAVA code for searching a given key in a squashed bucket

After computing the index `i` and the CHV value h, the search begins by considering the case of equality with K_1, which is done by comparing `keys[i]` with κ, and if these two are equal, `values[i]` is returned. In case of non-singleton bucket, the call k.`equals`(κ) invokes method `equals` of `Bucket2`. This virtual function call is made only if $h == h_1$. The search continues with a check whether a longer bucket resides in `keys[i]` by checking whether k is an `instanceof` class `Bucket2`, in which case we proceed to comparing κ with field K_2, etc.

The implementation of insertions and removals relied on a special case treatment of singleton buckets and dynamic dispatch of all other buckets.

A detailed analysis of the space overhead reduction is given in the appendix, e.g., it is shown that for $\mathcal{S}$-`HashMap` on HotSpot32, the expected bytes overhead per table entry is

$$24 - \left(55 + e^{-p} \cdot \left(64 + 40p + 20p^2 + 4p^3 + p^4/3 - p^5/15\right)\right)/p.$$

As it turns out, asymptotically, i.e., as p approaches infinity, the memory overheads of both $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` is the same as that of $\mathcal{L}$-hash, i.e., 24 (48) bytes per table entry on HotSpot32 (HotSpot64). The reason is that buckets of size $k > 6$ are not optimized in our implementation.

Nevertheless, in the typical range on HotSpot32, squashed hashing is even more memory efficient than fused hashing. $\mathcal{S}$-`HashSet` is particularly efficient, making an about two fold compaction in this range. It is also evident that both the *absolute* and *relative* savings in using squashed hashing are greater in the 64-bit memory model than in the 32-bits model.

6 Virtual Entries

Virtual entries enable read (and even write) access to the actual data (the "contained" portion in the Mitchell and Sevitsky taxonomy) in a data structure whose representation was compacted. Such access is required e.g., for in-order iteration over tree nodes, and for implementing methods in the `Map` interface (Fig. 6.1), which provide methods for the examination, and even change of ***(i)*** the set of all keys stored in the map, ***(ii)*** the multi-set of all values, and ***(iii)*** the set of all ⟨key, value⟩ pairs, nicknamed `Entry`.

The reference implementation of these methods makes use of a minimal collection data-structure containing objects that implement interface `Map.Entry`. Specifically, class `HashMap.Entry`, which defines entries in $\mathcal{L}$-`HashMap`, implements this interface. Function `entrySet()`, for example, returns an instance of `AbstractSet` wrapped around an iterator over the entire set of hash table entries.

```
public interface Map<K, V> {
  //...
  Set<K> keySet();
  Collection<V> values();
  Set<Map.Entry<K, V>> entrySet();
  //...
  interface Entry<K, V> {
    K getKey();
    V getValue()
    V setValue(V value);
    //...
  }
}
```

Fig. 6.1. Required methods for iteration over `Map` entries and interface `Map.Entry`

```
abstract class VirtualEntry<K, V>
 implements Map.Entry<K, V> {
  abstract
   VirtualEntry<K, V> next();
  protected abstract
   void setV(V v);
  @Override public final
    V setValue(V v) {
     V old = getValue();
     setV(v);
     return old;
  }
}
```

Fig. 6.2. Class `VirtualEntry`

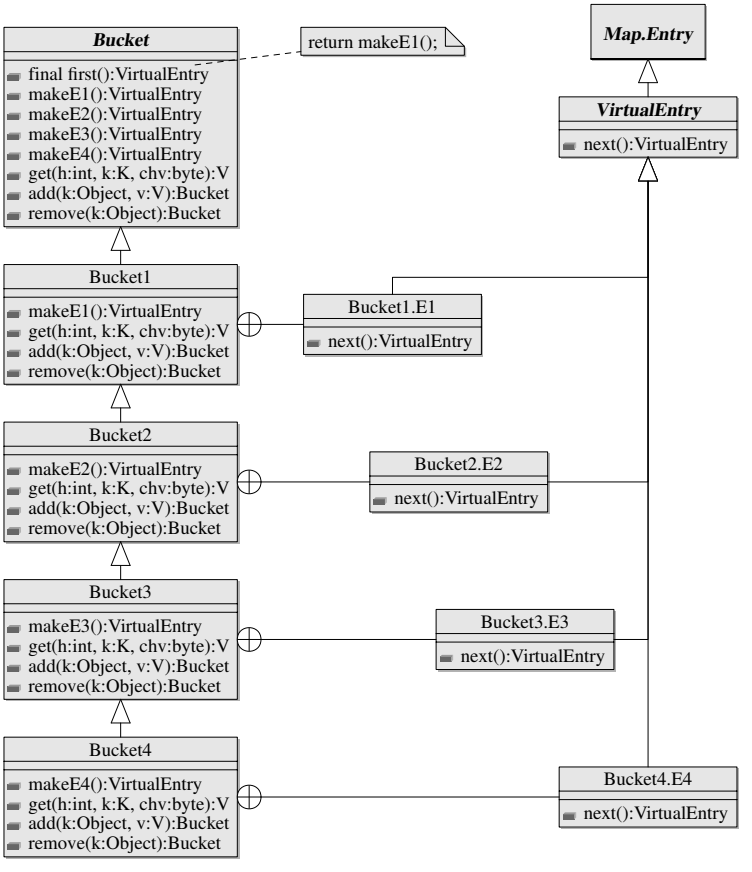


Fig. 6.3. A UML class diagram for virtual entry views on fused buckets

Class `VirtualEntry` (introduced in Fig. 6.2) presents a `Map.Entry` *view* on the fields defined e.g., in a fused bucket; where a fused bucket object typically offers a number of such views. Method `next()` in `VirtualEntry` returns the view of the next key-value pair, which is either in the same object or in a subsequent object.

The implementation of virtual entries for fused hash table includes a hierarchy of `VirtualEntry` subclasses `E1`, `E2`, `E3` and `E4`, which specialize the virtual entry concept for classes `Bucket1`, ..., `Bucket4`. Fig. 6.3 is a UML class diagram portraying the essentials.

Start with class `Bucket`; the class defines `add` and `remove` methods which are used for dynamic dispatch selection of the appropriate insertion and removal method based on the concrete bucket type. Similarly, for each of the virtual views `E1`, `E2`, `E3` and `E4`, this class defines factory methods, with default implementation returning `null`. The `final` method `first` in `Bucket` calls the first of these factory methods to return the first virtual entry stored in a fused bucket.

Then, each of `Bucket1`, ..., `Bucket4` ***(i)*** inherits the views of the class it `extends`, ***(ii)*** adds a view in its turn, and, ***(iii)*** overrides the corresponding

factory method defined in `Bucket` to return its view. For example, `Bucket2` ***(i)*** inherits the view `E1` from `Bucket1` ***(ii)*** defines the view `E2`, and, ***(iii)*** overrides the factory method `makeE2` to return an instance of this view. Class `E2`, an inner class of `Bucket2`, implements the `VirtualEntry` interface by direct access to the K_2, V_2 fields of `Bucket2`; method `next()` in `Bucket2.E2` calls the `makeE3()` factory method to generate the next view. this factory method returns `null` in this class, but overridden in `Bucket3` to return an instance of the view `Bucket3.E3`.

Function `next()` method in `Bucket4.E4` class is a bit special: if the `next` field is not `null` it returns the first view of the bucket object that follows.

The virtual entry views technique carries almost as is to squashed hashing: classes `Bucket`, `Bucket2`, ..., `Bucket6` and their inner virtual entry classes `E2`, ..., `E5` are obtained by almost mechanical application of the pattern by which each class ***(i)*** inherits the views of the class it `extends`, ***(ii)*** adds a view of its own, and, ***(iii)*** overrides the corresponding factory method defined in `Bucket`. Singleton buckets are special, since they are not represented by an object in squashed hashing. A virtual entry view of these is by class `HashMap.E1`, a non-static inner class of `HashMap`, which saves the table index passed to its constructor, as means for accessing later the singleton bucket residing at the $\texttt{i}^{th}$ position.

7 Time Performance of Fused and Squashed Hashing

The impact of fused and squashed hash table on application performance was benchmarked using three widely used standard suites: SPECjvm2008, SPECjbb2005 and DaCapo. To this end, we assembled two JRE versions; the first replacing $\mathcal{L}$-`HashMap` and $\mathcal{L}$-`HashSet` by $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet`, tailored for the HotSpot32 memory model; the second JRE version was prepared using $\mathcal{F}$-`HashMap` and $\mathcal{S}$-`HashSet` tailored for the HotSpot64 memory model.

Measurements were carried out on the 32-bit and 64-bit flavors of Linux Mint 11 (Katya) OS, installed on Intel Core i3 processor running at 2.93GHz clock rate and equipped with 2GB RAM. The benchmarked code was compiled with Eclipse Helios's compiler, and linked with JRE version 1.6.0_26-b03 with the respective flavor of HotSpot Server VM 20.1-b02, mixed-mode. To ensure a clean execution environment, the benchmarked machine was placed in a single user mode, with no network connection, and using text mode rather than GUI. Further, all but one core were disabled, and clock rate on that core was set in force mode to maximal value. We also made sure by manual inspection that no background applications were running.

Since certain benchmarks gave rise to great variety in timing results (even on identical, and as "clean" as possible settings), each benchmark was executed ten times with each of the three JRE versions (baseline, fused and squashed). Student's t-test was then employed to evaluate the statistical significance of the difference in running times; our reports include both the throughput change and significance levels, i.e., the α value. However, results in which the significance levels was less than 95%, i.e., $\alpha > 5\%$, are omitted.

SPECjvm2008 benchmark setup comprised `-Xms1500m -Xmx1500m` JVM arguments (i.e., 1.5GB initial and maximal heap) and non-default application arguments `-ict -ikv --peak -bt 1`. The results are given in Table 7.1. Of the eleven benchmarks in this suite, four did not exhibit any statistically significant change to the performance; five benchmarks exhibited a statistically significant change in only one memory model; the remaining two benchmarks exhibited significant throughput change in both memory models. Although some of the values in the table are positive, many show a small performance degradation. In the 32-bit version the average throughput change was −1.22% while the average throughput change in the 64-bit version was −2.10%.

Table 7.1. Throughput change and statistical significance in SPECjvm2008

	HotSpot32		**HotSpot64**	
	throughput change	α	*throughput change*	α
compiler	-2.37%	0.00%	-3.28%	0.00%
derbi	-1.94%	0.61%		
mpegaudio	0.49%	4.74%		
scimark.large			-5.12%	0.00%
scimark.small			1.88%	0.00%
serial	-1.61%	0.08%	-3.01%	0.27%
startup			-0.99%	4.38%
avg.	-1.22%		-2.10%	

Table 7.2. Throughput change and statistical significance in SPECjbb2005 warehouses benchmarks

	HotSpot32		**HotSpot64**	
	throughput change	α	*throughput change*	α
1 warehouse	-0.34%	1.98%	-2.26%	0.22%
2 warehouses	-1.25%	0.33%	-3.14%	0.02%
3 warehouses	-1.94%	0.15%	-4.35%	0.00%
4 warehouses	-1.87%	0.11%	-4.01%	0.00%
5 warehouses	-1.41%	3.01%	1.31%	4.05%
6 warehouses	22.74%	0.00%	1.82%	1.78%
7 warehouses	1.74%	0.00%	5.82%	0.00%
8 warehouses	6.80%	0.00%	24.17%	0.00%
avg.	3.06%		2.42%	

On SPECjbb2005, the JVM arguments were `-Xms256m -Xmx256m`, while the non-default application arguments were `input.deterministic_random_seed=true`. Results are presented in Table 7.2. Evidently, all measured value were statistically significant. Negative impact on throughput typifies the smaller numbers of warehouses, but positive impact is witnessed in the larger number of warehouses. Overall, in the 32-bit memory model the *average* throughput change was 3.06%; in the 64-bit version it was 2.42%.

The JVM arguments for the DaCapo benchmark were `-Xms1500m -Xmx1500m` while the non-default application arguments were `--no-validation -C -t 1`. A few benchmarks could not be applied to our re-implementation of hash tables. The reason is that these benchmarks relied on a stored serialized version of the collection under test. The benchmarks actually used were therefore: avrora, batik, eclipse, h2, jython, luindex, lusearch, pmd, sunflow, tomcat and xalan. All statistically significant results are presented in Table 7.3. In the 32-bit version the average speedup was -0.74% while the average speedup in the 64-bit version was 4.71%.

We conclude that our proposed implementations remain within practical runtimes, imposing in some cases speedup to the JVM compared to the base implementation, while imposing significant memory overhead reduction.

Table 7.3. Throughput change and statistical significance in DaCapo

	HotSpot32		HotSpot64	
	speedup	α	*speedup*	α
batik	5.30%	2.18%		
h2	1.65%	3.55%		
jython			1.17%	1.26%
luindex	-17.32%	1.30%	14.55%	0.23%
pmd	8.65%	0.34%		
xalan	-1.92%	0.00%	-1.59%	0.09%
avg.	-0.74%		4.71%	

Slowdowns, when they occur, do not come as a surprise as the common usage of hash tables is as tiny (less than 16 entries) collections[6] . Naturally, our data-structures non-trivial encoding requires a more sophisticated decoding. Some operations are expected to noticeably slow down, i.e.: iterations and removals, compared to the almost trivial baseline implementation of them.

Although guided by some benchmarking, the majority of the code in our implementation was not hand optimized to achieve the ultimate time performance. It is desirable of course to make this possible.

8 Compaction of Balanced Binary Tree Nodes

This section describes the two schemes for compact representation of tree nodes of `TreeMap` (and `TreeSet`), i.e., class `TreeMap.Entry` (Fig. 1.1(a)): *fused binary tree* achieves this compaction with null-pointer and boolean eliminations; *squashed binary tree* consolidates all fields in `TreeMap.Entry`, replacing pointers by integers. The memory saving that these achieve are as reported above in Table 1.3.

Fusion. Field `color` is an obvious candidate for boolean elimination. Also, since half of the children edges in binary trees (fields `left` and `right` in `TreeMap.Entry`) are `null`, null-pointer-elimination is applicable to `left` and `right`. Fig. 8.1 shows how these techniques can be used for the compaction of leaves.

Class `Node` (implementing interface `VNode`) is the base class of all specialized tree node classes; it defines `key`, `value` and `parent` fields, just like `TreeMap.Entry`, except that the parent is necessarily an internal node (class `Internal`). Fields `left`, `right` are represented as `abstract` getter functions; null elimination of these fields is by subclass `Leaf` overriding these functions to return `null`. Field `color` is modeled as abstract getter and setter methods. The contract of the setter `color(c)` is that if `c` is different from the current node's color, it returns a new node which is identical, except for the color. The setters' implementation in class `Leaf`, creates either a `RLeaf` or `BLeaf` object as necessary.

Classes `RLeaf` and `BLeaf` have only three data fields, all of which are pointers. The object size of these classes is thus 24 bytes (with four bytes wasted on

[6] `http://mail.openjdk.java.net/pipermail/core-libs-dev/2009-July/001969.html`

```
interface VNode<K, V>
 extends Map.Entry<K, V> {
  public boolean color();
  public VNode<K, V> parent();
  public VNode<K, V> left();
  public VNode<K, V> right();
  //...
}
abstract static class Node<K, V>
 implements VNode<K, V> {
  final static boolean //colors
   BLACK = true,
   RED = !BLACK;
  K key; V value; //contents
  Internal<K, V> parent; //topology
  VNode(K k, V v, Internal<K, V> p) {
   key = k;
   value = v;
   parent = p;
  }
  public final VNode<K, V> parent() {
   return parent;
  }
  public abstract Node<K, V>
   color(boolean c);
  //...
}
abstract static class Leaf<K, V>
 extends Node<K, V> {
  Leaf(K k, V v, Internal<K, V> p) {
   super(k,v,p);
  }
  final Node<K, V> left() {
   return null;
  }
  final Node<K, V> right() {
   return null;
  }
  final Leaf<K, V> color(boolean c) {
   return c==color() ? this : make(c);
  }
  private Leaf<K, V> make(boolean c) {
   Leaf<K, V> l = c==BLACK
      ? new BLeaf<K, V>(this)
      : new RLeaf<K, V>(this);
   //...
   return l;
  }
  //...
}
static final class RLeaf<K, V>
 extends Leaf<K, V> {
  RLeaf(Node<K, V> l) {
   super(l.key, l.value, l.parent);
  }
  @Override final public boolean color() {
   return RED;
  }
  //...
}
static final class BLeaf<K, V>
 extends Leaf<K, V> {
  BLeaf(Node<K, V> l) {
   super(l.key, l.value, l.parent);
  }
  @Override final public boolean color() {
   return BLACK;
  }
  //...
}
```

Fig. 8.1. Employing null pointer elimination and boolean elimination for the compaction of leaf nodes in red-black binary search tree

alignment) on HotSpot32 and 40 bytes (with no alignments waste) on HotSpot64. The size of the `TreeSet` version of these classes is 16 bytes on HotSpot32, and 32 bytes on HotSpot64.

We empirically found that $\approx 42.8\%$ of nodes in a red-black tree are leaves, and that this ratio is independent of tree size, nor of tree creation order.[7] Employing classes `RLeaf` and `BLeaf`, in an implementation of `TreeMap` reduces overhead from 24 to 20.6 bytes on HotSpot32 (respectively 48 to 37.7 on HotSpot64) which amounts to 14% (21%) savings. With `TreeSet` the respective reductions are 28 to 21.1 and 56 to 42.3 (both 24% saving).

Since each of the tree's nodes is "owned" by its parent, it makes sense to apply fusion in tree nodes, just as we did for lists. The difficulty is in dealing with the very many cases that could occur: Depth-ℓ fusion may entail a specialized class for the $O(2^\ell)$ trees of this depth. Attentions is therefore restricted to fusion of nodes with their leaves (i.e., $\ell = 2$), distinguishing between three different cases: *(i)* internal nodes which are parents to two leaves, *(ii)* internal nodes which are

[7] This high value fraction is not accidental; similar fractions occur in e.g., AVL trees. We can in fact analytically prove that about one quarter of the nodes are leaves in a *random unbalanced* binary tree; balancing leads to increasing the number of leaves.

parents to a single leaf, the other child being `null`, and *(iii)* nodes in which one of the children is a leaf and the other is a non-leaf non-`null` node, which we will ignore.

The characteristics of a red-black tree limit the variety of colors in cases (i) and (ii). In (i), if the parent is RED, then children are both BLACK (if the parent is BLACK then colors of children may be of any color). In (ii), if the parent is BLACK then its leaf child must be RED. Five different concrete types of nodes are thus defined: `ParentLeftLeaf`, `ParentRightLeaf`, `ParentLeavesRBB`, `ParentLeavesBBB`, and, `ParentLeavesBRR`.

As shown in Fig. 8.2, the first two classes have five pointer fields, Fields `parent`, `key` and `value` are inherited from the superclass `Node`, while two additional pointers, `keyChild` and `valueChild` are defined in `ParentLeaf` which is the `abstract` superclass of both `ParentLeftLeaf` and `ParentRightLeaf`.

```
abstract static class
 ParentLeaf<K, V>
  extends Node<K, V> {
   protected K keyChild;
   protected V valueChild;
   //methods and inner classes...
}
static final class
 ParentLeftLeaf<K, V>
  extends ParentLeaf<K, V> {
   //methods and inner classes...
}
static final class
 ParentRightLeaf<K, V>
  extends ParentLeaf<K, V> {
  //methods and inner classes...
}
abstract static class
 ParentLeaves<K, V>
  extends ParentLeaf<K, V> {
   protected final K keyRightLeaf;
   protected V valueRightLeaf;
  //methods and inner classes...
}
static final class
 ParentLeavesRBB<K, V>
  extends ParentLeaves<K, V> {
  //methods and inner classes...
}
static final class
 ParentLeavesBBB<K, V>
  extends ParentLeaves<K, V> {
  //methods and inner classes...
}
static final class
 ParentLeavesBRR<K, V>
  extends ParentLeaves<K, V> {
  //methods and inner classes...
}
```

Fig. 8.2. Classes for fusion of internal tree nodes with their leaf children

The footprint of `ParentLeftLeaf` and `ParentRightLeaf` classes for `TreeMap` is $8 + 5 \cdot 4 = 28$ bytes, which are up-aligned to 32 bytes (on HotSpot32). Class `ParentLeaves` adds two more pointers, making a 40 bytes footprint for each of its three subclasses on HotSpot32.

Empirically we found that 14% of the nodes in a red-black tree have a single leaf child, while 9% of the nodes have two leaves as their children. Then, $2 \cdot 14\% = 28\%$ of the nodes consume $32/2 = 16$ bytes each, while $3 \cdot 9\% = 27\%$ of the nodes consume $40/3 = 13.3$ bytes each. Assuming that no compression is done for the other nodes, we obtain 14.5 bytes of overhead per node, achieving 40% savings in overhead per node, just by using leaf level fusion. If the remaining leaves are represented as in Fig. 8.1, then saving increases to 43% for $\mathcal{F}$-`TreeMap` on HotSpot32.

Our implementation of fused binary trees goes further, employing boolean elimination to class `TreeMap.Entry` (Fig. 1.1(a)) which is used for all other

```
abstract static class Internal<K, V>
 extends Node<K, V> {
  Node<K, V> left, right;
  //...
  @Override final public
   VNode<K, V> left() {
    return left;
  }
  @Override final public
    VNode<K, V> right() {
      return right;
  }
  @Override final Internal<K, V>
    color(boolean c) {
     return c==color()
           ? this
           : make(c);
  }
  private
   Internal<K, V> make(boolean c) {
    Internal<K, V> n = c==BLACK
          ? new BInternal<K, V>(this)
          : new RInternal<K, V>(this);
     //...
     return n;
   }
   //...
}
final static class RInternal<K, V>
  extends Internal<K, V> {
   //...
   @Override final public boolean color() {
    return RED;
   }
}
final static class BInternal<K, V>
 extends Internal<K, V> {
  //...
   @Override final public boolean color() {
    return BLACK;
   }
}
```

Fig. 8.3. Abstract class `Internal` and boolean elimination in specialized internal nodes

kinds of tree nodes (internal nodes). (Described in Fig. 8.3.) Such elimination capitalizes the memory reduction for the other $\mathcal{F}$-tree types besides $\mathcal{F}$-`TreeMap` on HotSpot32, which due to alignment waste does not benefit from it. Combined with the above techniques we achieve the following overhead savings: *(i)* 59% for `TreeSet` on HotSpot32, *(ii)* 55% for `TreeMap` on HotSpot64, and *(iii)* 61% for `TreeSet` on HotSpot64.

Not surprisingly, with nine different concrete classes for tree nodes, coding was not easy. Difficulties include the fact that tree updates may turn internal nodes into leaves and vice versa, and that rotations may change the tree topology. Removals were particularly challenging as they may initiate any number of tree rotations. As with hashing, dynamic dispatch was employed for abstracting over the variety of node types, and virtual entries were used for iterating over the tree nodes. Initial benchmarking results indicate that this abstraction layer lead to 30-50% slowdown.

Full Field Consolidation. Squashing encodes the entire `TreeMap<K,V>` data structure without using *any* small objects. Instead, six tree-global arrays, `K[] key`, `V value[]`, `boolean[] color`, `int[] left`, `int[] right`, and `int[] parent` consolidate the fields of all tree nodes; node i is then the $\mathtt{i}^{th}$ location in all of these arrays, while pointers are replaced by array indices. This consolidation eliminates both headers of all small objects, and alignment waste incurred to the byte sized `color` field. When the arrays are fully occupied, a node overhead is reduced from 32 to 21 bytes on HotSpot32, and from 64 bytes to 20 bytes on HotSpot64. On both memory models overhead is 13 bytes, and we achieve the following overhead savings: *(i)* 46% for `TreeMap` on HotSpot32, *(ii)* 54% for `TreeSet` on HotSpot32, *(iii)* 73% for `TreeMap` on HotSpot64, and *(iv)* 77% for `TreeSet` on HotSpot64. Implementation, as expected, was almost mechanical; initial benchmarking results indicate that the performance of the squashed tree implementation is comparable to the baseline, and sometimes even slightly faster.

Note that squashed trees come at the cost of shifting memory management duties from the JVM back to the programmer. To avoid costly reallocations, array should include sufficient slack. Still, a generous 50% slack places field consolidation behind fusion.

It is feasible to implement a version of squashed trees, in which the arrays' type changes by the current size of the collection: `byte[]` arrays, then `short[]` arrays and finally `int[]` arrays. The expected saving increases for small collections as summarized in Table 8.1. Table ignores *instance-memory* waste (See 3) which may be slightly more significant for smaller objects. Accounting for this waste, the expected addition is of two bytes per key for a collection of size 50, and decreases rapidly as size increases.

Table 8.1. Computed saving in memory overhead per tree entry due to the use of full consolidation using different types of indices in the implementation of `TreeMap` and `TreeSet` for HotSpot32 and HotSpot64

	HotSpot32			**HotSpot64**		
	int	*short*	*byte*	*int*	*short*	*byte*
`TreeMap`	46%	71%	83%	73%	85%	92%
`TreeSet`	54%	75%	86%	77%	88%	93%

9 Further Research

This research raises a number of interesting questions. First, it is important and interesting to understand better the domain of tiny collections, of say up to say 16 entries, as their relative overhead is more significant. Reducing the overhead of these seems more challenging, especially in adhering to the very general `Map` interface. It would be useful to make estimates on abundance of tiny collections in large programs and the manner in which they are used, with the conjecture that a frugal yet less general implementation of these would be worthwhile.

Second, as we have seen in this work the variety of the user-level compaction algorithms are not always easy to employ. A software framework or better yet, automatic tools that abstract over encoding issues would make our findings more accessible. It is crucial for such a framework to be able to produce code for both (say) `TreeMap` and `TreeSet` without code duplication. The virtual entries technique presented in Sect. 6 may serve as a starting point, but other directions may include the use of aspects or more sophisticated generics.

Third, we are intrigued by the fact that despite fewer dereferencing operations, $\mathcal{F}$-hash and $\mathcal{S}$-hash were not significantly faster than $\mathcal{L}$-hash. Micro-benchmark of individual operations should not only clarify this point, but also make room for systematic hand- and later automatic- optimization of these.

Finally, we draw attention to the problems of memory profiling, a meaningful and precise definition of the notion of "footprint" of an application, and its impact on time. These issues are illusive since the "footprint" changes in the course of computation, and the memory consumption curve may depend on garbage collection cycles, which in general are not deterministic, yet may depend on the allocation of physical memory and other factors.

References

1. Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V.M., Stichnoth, J.M.: Fast, effective code generation in a just-in-time Java compiler. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2008), Tucson, Arizona, June 7-13. ACM Press (2008)
2. Alon, N., Dietzfelbinger, M., Miltersen, P.B., Petrank, E., Tardos, G.: Linear hash functions. J. ACM 46 (September 1999)
3. Arbitman, Y., Naor, M., Segev, G.: Backyard Cuckoo hashing: Constant worst-case operations with a succinct representation. In: Proc. of the 51st IEEE Annual Symp. on Foundation of Comp. Sci. (FOCS 2010), Las Vegas, Navada, October 23-26. IEEE Computer Society Press (2010)
4. Bacon, D.F., Fink, S.J., Grove, D.: Space and time efficient implementation of the Java object model. In: Proc. of the 17th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2002), Seattle, Washington, November 4-8 (2002); ACM SIGPLAN Notices 37(11)
5. Caromel, D., Reynders, J., Philippsen, M.: Benchmarking Java against C and Fortran for scientific applications. In: Thomas, D. (ed.) Proc. of the 20th Euro. Conf. on OO Prog. (ECOOP 2006), Nantes, France, July 3-7. LNCS, vol. 4067. Springer (2006)
6. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 1998), Vancouver, British Columbia, Canada, October 18-22 (1998); ACM SIGPLAN Notices 33(10)
7. Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Wolczko, M.: Compiling Java just in time. IEEE Micro 17(3) (May/June 1998)
8. Cranor, L.F., Wright, R.N.: Influencing software usage. In: Proc. of the 10th Conference on Computers, Freedom and Privacy (CFP 2000). ACM (2000)
9. Dufour, B., Ryder, B.G., Sevitsky, G.: A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In: Proc. of the 16th ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (FSE 2008), Atlanta, Georgia, November 9-14. ACM Press (2008)
10. Eckel, N., Gil, J.: Empirical Study of Object-Layout Strategies and Optimization Techniques. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 394–421. Springer, Heidelberg (2000)
11. Feller, W.: An Introduction to Probability Theory and Its Applications, vol. I. Wiley (1968)
12. Gil, Y., Lenz, K., Shimron, Y.: A microbenchmark case study and lessons learned. In: Proc. of the 5th International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (VMIL 2011). ACM (2011)
13. Hsieh, C.H.A., Conte, M.T., Johnson, T.L., Gyllenhaal, J.C., Hwu, W.M.W.: Compilers for improved Java performance. Computer 30(6) (June 1997)
14. Kawachiya, K., Ogata, K., Onodera, T.: Analysis and reduction of memory inefficiencies in Java strings. In: Harris, G.E. (ed.) Proc. of the 23rd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2008), Nashville, Tennessee, October 19-23. ACM (2008)
15. Kotzmann, T., Wimmer, C., Mossenbock, H., Rodriguez, T., Russell, K., Cox, D.: Design of the Java HotSpot client compiler for Java 6. ACM Trans. Prog. Lang. Syst. 5(1) (May 2008)

16. Maxwell, E.K., Back, G., Ramakrishnan, N.: Diagnosing memory leaks using graph mining on heap dumps. In: Proc. of the 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2010), Washington, DC, July 25-28. ACM Press (2010)
17. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to Java runtime bloat. IEEE Software 27(1) (2010)
18. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: Gabriel, R.P., Bacon, D. (eds.) Proc. of the 22nd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2007), Montreal, Quebec, Canada, October 21-25. ACM Press (2007)
19. Moreira, J.E., Midkiff, S.P., Gupta, M.: A comparison of Java, C/C++, and FORTRAN for numerical computing. IEEE Antennas and Propagation Magazine 40(5), 102–105 (1998)
20. Novark, G., Berger, E.D., Zorn, B.G.: Efficiently and precisely locating memory leaks and bloat. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2009)
21. Ogata, K., Mikurube, D., Kawachiya, K., Onodera, T.: A study of Java's non-Java memory. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proc. of the 25th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2010), Reno/Tahoe, Nevada, USA, October 17-21. ACM (2010)
22. Paleczny, M., Vick, C., Click, C.: The Java Hotspot server compiler. In: Proc. of the Java Virtual Machine Research and Technology Symposium (JVM 2001), Monterey, California, April 23-24. USENIX C++ Technical Conf. Proc. (2001)
23. Reiss, S.P.: Visualizing the Java heap. In: Proc. of the 32nd Int. Conf. on Soft. Eng. (ICSE 2010), Cape Town, South Africa, May 2-8. ACM (2010)
24. Shacham, O., Vechev, M., Yahav, E.: Chameleon: Adaptive selection of collections. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2009), Dublin, Ireland, June 15-20. ACM Press (2009)
25. Venstermans, K., Eeckhout, L., Bosschere, K.D.: Java object header elimination for reduced memory consumption in 64-bit virtual machines. TACO 4(3) (2007)
26. Wimmer, C.: Automatic Object Inlining in a Java Virtual Machine. PhD thesis, Institute for System Software, Johannes Kepler University Linz (2008)
27. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: Profiling copies to find runtime bloat. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2009)
28. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: Proc. of the 18th ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (FSE 2010), Santa Fe, New Mexico, November 7-11. ACM Press (2010)
29. Xu, G., Rountev, A.: Precise memory leak detection for Java software using container profiling. In: Proc. of the 30th Int. Conf. on Soft. Eng. (ICSE 2008), Leipzig, Germany, May 10-18. ACM (2008)
30. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2010), Toronto, Canada, June 5-10. ACM Press (2010)

Enhancing JavaScript with Transactions

Mohan Dhawan[1], Chung-chieh Shan[2], and Vinod Ganapathy[1]

[1] Rutgers University
{mdhawan,vinodg}@cs.rutgers.edu
[2] University of Tsukuba
ccshan@post.harvard.edu

Abstract. Transcript is a system that enhances JavaScript with support for transactions. Hosting Web applications can use transactions to demarcate regions that contain untrusted guest code. Actions performed within a transaction are logged and considered speculative until they are examined by the host and committed. Uncommitted actions simply do not take and cannot affect the host in any way. Transcript therefore provides hosting Web applications with powerful mechanisms to understand the behavior of untrusted guests, mediate their actions and also cleanly recover from the effects of security-violating guest code.

This paper describes the design of Transcript and its implementation in Firefox. Our exposition focuses on the novel features introduced by Transcript to support transactions, including a suspend/resume mechanism for JavaScript and support for speculative DOM updates. Our evaluation presents case studies showing that Transcript can be used to enforce powerful security policies on untrusted JavaScript code, and reports its performance on real-world applications and microbenchmarks.

1 Introduction

It is now common for Web applications (*host*) to include untrusted, third-party JavaScript code (*guest*) of arbitrary provenance in the form of advertisements, libraries and widgets. Despite advances in language and browser technology, JavaScript still lacks mechanisms that enable Web application developers to debug and understand the behavior of third-party guest code. Using existing reflection techniques in JavaScript, the host cannot attribute changes in the JavaScript heap and the DOM to specific guests. Further, fine-grained context about a guest's interaction with the host's DOM and network is not supported. For example, the host cannot inspect the behavior of guest code under specific cookie values or decide whether to allow network requests by the guests.

This paper proposes to enhance the JavaScript language with builtin support for introspection of third-party guest code. The main idea is to extend JavaScript with a new `transaction` construct, within which hosts can speculatively execute guest code containing arbitrary JavaScript constructs. In addition to enforcing security policies on guests, a `transaction` would allow hosts to *cleanly recover from policy-violating actions of guest code*. When a host detects an offending guest, it simply chooses not to commit the transaction corresponding to the guest. Such an approach neutralizes any data and DOM modifications initiated earlier by the guest, without having to undo them

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 383–408, 2012.

```
1  <script type="text/javascript">
2  var editor = new Editor(); initialize(editor);
3  var builtins = [], tocommit = true;
4  for(var prop in Editor.prototype) builtins[prop] = prop;
5  var tx = transaction {   Guest code: Lines 6–9
6      Editor.prototype.getKeywords = function(content) {...}
       ...
7      var elem = document.getElementById("editBox");
8      elem.addEventListener("mouseover", displayAds, false);
       ...
9      document.write('<div style="opacity:0.0; z-index:0; ... size/loc params">
           <a href="http://evil.com"> Evil Link </a></div>');
10 };
11 tocommit = gotoIblock(tx);   Implements the host's security policies
12 if (tocommit) tx.commit();
13 ... /* rest of the Host Web application's code */
14 </script>
```

Fig. 1. Motivating example. This example shows how a host can mediate an untrusted guest (lines 6–9). The introspection block (invoked in line 11) enforces the host's security policies (see Figure 2) on the actions performed by the guest.

explicitly. The introspection mechanism (`transaction`) is built within the JavaScript language itself, thereby allowing guest code to contain arbitrary JavaScript constructs (unlike contemporary techniques [13,18,36,29,31]).

Let us consider an example of a Web-based word processor that hosts a third-party widget to display advertisements (see Figure 1). During an editing session, this widget scans the document for specific keywords and displays advertisements relevant to the text that the user has entered. Such a widget may modify the host in several ways to achieve its functionality, *e.g.,* it could install event handlers to display advertisements when the user places the mouse over specific phrases in the text. However, as an untrusted guest, this widget may also contain malicious functionality, *e.g.,* it could implement a clickjacking-style attack by overlaying the editor with transparent HTML elements pointing to malicious sites.

Traditional reference monitors [16], which mediate the action of guest code as it executes, can detect and prevent such attacks. However, such reference monitors typically only enforce access control policies, and would have let the guest modify the host's heap and DOM (such as to install innocuous event handlers) until the attack is detected. When such a reference monitor reports an attack, the end-user faces one of two unpalatable options: (a) close the editing session and start afresh; or (b) continue with the tainted editing session. In the former case, the end-user loses unsaved work. In the latter case, the editing session is subject to the unknown and possibly undesirable effects of the heap and DOM changes that the widget initiated before being flagged as malicious. In our example, the event handlers registered by the malicious widget may also implement undesirable functionality and should be removed when the widget's clickjacking attempt is detected.

Speculative execution allows hosts to introspect all actions of untrusted guest code. In our example, the host speculatively executes the untrusted widget by enclosing it in a transaction. When the attack is detected, the host simply discards all changes initiated by the widget. The end-user can proceed with the editing session without losing unsaved work, and with the assurance that the host is unaffected by the malicious widget.

This paper describes the Transcript system, that has the following novel features:

(**1**) **JavaScript Transactions.** Transcript allows hosting Web applications to speculatively execute guests by enclosing them in transactions. Transcript maintains *read and write sets* for each transaction to record the objects that are accessed and modified by the corresponding guest. These sets are exposed as properties of a *transaction object* in JavaScript. Changes to a JavaScript object made by the guest are visible within the transaction, but any accesses to that object from code outside the transaction return the unmodified object. The host can inspect such speculative changes made by the guest and determine whether they conform to its security policies. The host must explicitly commit these changes in order for them to take effect; uncommitted changes simply do not take and need not be undone explicitly.

(**2**) **Transaction Suspend/Resume.** Guest code may attempt operations outside the purview of the JavaScript interpreter. In a browser, these *external operations* include AJAX calls that send network requests, such as `XMLHttpRequest`. Transcript introduces a *suspend and resume* mechanism that affords unprecedented flexibility to mediate external operations. Whenever a guest attempts an external operation, Transcript suspends it and passes control to the host. Depending on its security policy, the host can perform the action on behalf of the guest, perform a different action unbeknownst to the guest, or buffer up and simulate the action, before resuming this or another suspended transaction.

(**3**) **Speculative DOM Updates.** Because JavaScript interacts heavily with the DOM, Transcript provides a speculative DOM subsystem, which ensures that DOM changes requested by a guest will also be speculative. Together with support for JavaScript transactions, Transcript's DOM subsystem allows hosts to cleanly recover from attacks by malicious guests.

Transcript provides these features without restricting or modifying guest code in any way. This allows reference monitors based on Transcript to mediate the actions of legacy libraries and applications that contain constructs that are often disallowed in safe JavaScript subsets [13,18,36,29,31] (*e.g.,* `eval`, `this` and `with`).

In the rest of the paper, we discuss the design, implementation and evaluation of Transcript.

2 Overview of Transcript

Transcript enables hosts to understand the behavior of untrusted guests, detect attacks by malicious guests and recover from them, and perform forensic analysis. We briefly discuss Transcript's utility and then provide an overview of its functionality for confining a malicious guest.

(**1**) **Understanding Guest Code.** Analysis of third-party JavaScript code is often hard due to code obfuscation. Using Transcript, a host can set watchpoints on objects of interest. Coupled with suspend/resume, it is possible to perform a fine grained debug analysis by inspecting the read/write sets on every guest initiated object read/write and method invocation. Transcript's speculative execution provides an ideal platform for concolic unit testing [44,20] of guests. For example, using Transcript, a host can test a guest's behavior under different values of domain cookies.

(**2**) **Confining Malicious Guests.** Transcript's speculative execution permits buffering of network I/O and writing to a speculative DOM, thereby allowing unprecedented flexibility in confining untrusted guest code. For example, to prevent clickjacking-style attacks, the host can simply discard guest's modifications to the speculative DOM.

(**3**) **Forensic Analysis.** Since Transcript suspends on external and user-defined operations, the suspend/resume mechanism is an effective tool for forensic analysis of a suspected vulnerability exploited by the guest. For example, code-injection attacks using DOM or host APIs [4] can be analyzed by observing the sequence of suspend calls and their arguments.

Transcript in Action. We illustrate Transcript's ability to confine untrusted guests by further elaborating on the example introduced in Section 1. Suppose that the word processor hosts the untrusted widget using a `<script>` tag, as follows: `<script src="http://untrusted.com/guest.js">`. In Figure 1, lines 6–9 show a snippet from `guest.js`, which displays advertisements relevant to keywords entered in the editor. Line 6 registers a function to scan for keywords in the editor window by adding it to the prototype of the `Editor` object. Lines 7 and 8 show the widget registering an event handler to display advertisements on certain mouse events. While lines 6–8 encode the core functionality related to displaying advertisements, line 9 implements a clickjacking-style attack by creating a transparent `<div>` element, placed suitably on the editor with a link to an evil URL.

When hosting such a guest, the word processor can protect itself from attacks by defining and enforcing a suitable set of security policies. These may include policies to prevent prototype hijacks [41], clickjacking-style attacks, drive-by downloads, stealing cookies, snooping on keystrokes, *etc*. Further, if an attack is detected and prevented, it should not adversely affect normal operation of the word processor. We now illustrate how the word processor can use Transcript to achieve such protection and cleanly recover from attempted attacks.

The host protects itself by embedding the guest within a `transaction` construct (line 5, Figure 1) and specifies its security policy (lines D–O, Figure 2). When the transaction executes, Transcript records all reads and writes to JavaScript objects in per-transaction read/write sets. Any attempts by the guest to modify the host's JavaScript objects (*e.g.,* on line 6, Figure 1) are speculative; *i.e.,* these changes are visible only to the guest itself and do not modify the host's view of the JavaScript heap. To ensure that DOM modifications by the guest are also speculative, Transcript's DOM subsystem clones the host's DOM at the start of the transaction and resolves all references to DOM objects in a transaction to the cloned DOM. Thus, references to `document` within the guest resolve to the cloned DOM.

```
A   do {  Function gotoIblock implements the host's introspection block: Lines A–R
B       var arg = tx.getArgs();   var obj = tx.getObject();
C       var rs = tx.getReadSet(); var ws = tx.getWriteSet();
D       for(var i in builtins) {
E           if (ws.checkMembership(Editor.prototype, builtins[i])) tocommit = false;
F       } ... /* definition of 'IsClickJacked' to go here */
G       if (IsClickJacked(tx.getTxDocument())) tocommit = false;
H       ... /* more policy checks go here */  inlined code from libTranscript: Lines I–O
I       switch(tx.getCause()) {
J           case "addEventListener":
K               var txHandler = MakeTxHandler(arg[1]);
L               obj.addEventListener(arg[0], txHandler, arg[2]); break;
M           case "write": WriteToTxDOM(obj, arg[0]); break; ... /* more cases */
N           default: break;
O       };
P       tx = tx.resume();
Q   } while(tx.isSuspended());
R   return tocommit;
```

Fig. 2. An iblock. An iblock consists of two parts: a host-specific part, which encodes the host's policies to confine the guest (lines D–H), and a mandatory part, which contains functionality that is generic to all hosts (lines I–O).

When the guest performs DOM operations, such as those on lines 7–9, and other external operations, such as `XMLHttpRequest`, Transcript *suspends* the transaction and passes control to the host. This situation is akin to a system call in a user-space program causing a trap into the operating system. Suspension allows the host to mediate external operations as soon as the guest attempts them. When a transaction suspends or completes execution, Transcript creates a *transaction object* in JavaScript to denote the completed or suspended transaction. In Figure 1, the variable `tx` refers to the transaction object. Transcript then passes control to the host at the program point that syntactically follows the transaction. There, the host implements an *introspection block* (or *iblock*) to enforce its security policy and perform operations on behalf of a suspended transaction.

Transaction Objects. A transaction object records the state of a suspended or completed transaction. It stores the read and write sets of the transaction and the list of activation records on the call stack of the transaction when it was suspended. It provides builtin methods, such as `getReadSet` and `getWriteSet` shown in Figure 2, that the host can invoke to access read and write sets, observe the actions of the guest, and make policy decisions.

When a guest tries to perform an external operation and thus suspends, the resulting transaction object contains arguments passed to the operation. For example, a transaction that suspends due to an attempt to modify the DOM, such as the call `document.write` on line 9, will contain the DOM object referenced in the operation (`document`), the name of the method that caused the suspension (`write`), and the arguments passed to the method. (Recall that Transcript's DOM subsystem ensures that `document` referenced within the transaction will point to the cloned DOM.) The host can access these arguments using builtin methods of the transaction object, such as `getArgs`, `getObject` and `getCause`. Depending on its policy, the host can either perform the operation on behalf of the guest, simulate the effect of performing it, defer the operation for later, or not perform it at all.

The host can resume a suspended transaction using the transaction object's builtin `resume` method. Transcript then uses the activation records stored in the transaction object to restore the call stack, and resumes control at the program point following the instruction that caused the transaction to suspend (akin to resumption of program execution following a system call). Transactions can suspend an arbitrary number of times until they complete execution. The builtin `isSuspended` method determines whether the transaction is suspended or has completed.

A completed transaction can be committed using the builtin `commit` method. This method copies the contents of the write set to the corresponding objects on the host's heap, thereby publishing the changes made by the guest. It also synchronizes the host's DOM with the cloned version that contains any DOM modifications made by the guest. A completed transaction's call stack is empty, so attempts to resume a completed transaction will have no effect. Note that Transcript does not define an explicit `abort` operation. This is because the host can simply discard changes made by a transaction by choosing not to commit them. If the transaction object is not referenced anymore, it will be garbage-collected.

Introspection Blocks. When a transaction suspends or completes, Transcript passes control to the instruction that syntactically follows the transaction in the code of the host. At this point, the host can check the guest's actions by encoding its security policies in an *iblock*. The iblock in Figure 2 has two logical parts: a *host-specific part* that encodes host's policies (lines D–H), and a *mandatory part* that performs operations on behalf of suspended guests (lines I–O). The iblock in Figure 2 illustrates two policies:

(**1**) Lines D–E detect prototype hijacking attempts on the `Editor` object. To do so, they check the transaction's write set for attempted redefinitions of builtin methods and fields of the `Editor` object.

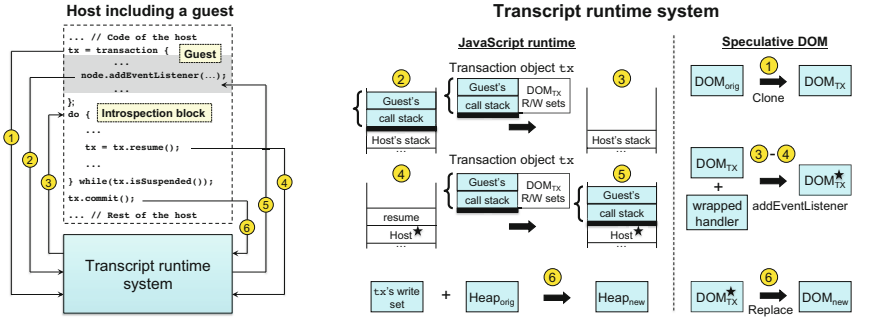
(**2**) Line G detects clickjacking-style attempts by checking the DOM for the presence of any transparent HTML elements introduced by the guest. (The body of `IsClickJacked`, which implements the check, is omitted for brevity).

The body of the `switch` statement encodes the mandatory part of the iblock and implements two key functionalities, which are further explained in Section 3.1:

(**1**) Lines J–L in Figure 2 create and attach an event handler to the cloned DOM when the guest suspends on line 8 in Figure 1. The `MakeTxHandler` function creates a new *wrapped* handler, by enclosing the guest's event handler (`displayAds`) within a `transaction` construct. Doing so ensures that the execution of any event handlers registered by the guest is also speculative, and mediated by the host's security policies. The iblock then attaches the event handler to the corresponding element (`elem`) in the cloned DOM.

(**2**) Line M in Figure 2 speculatively executes the DOM modifications requested when the guest suspends on line 9 in Figure 1. The `WriteToTxDOM` function invokes the `write` call on `obj`, which points to the `document` object in the cloned DOM.

If a transaction does not commit because of a policy violation, the host's DOM and JavaScript objects will remain unaffected by the guest's modifications. For instance,



(a) Locations of traps/returns to/from Transcript.

(b) Corresponding actions within the Transcript runtime system for a trap/return.

Fig. 3. Workflow of a Transcript-enhanced host. Part (a) of the figure shows a host enclosing a guest within a transaction and an inlined introspection block, while part (b) shows the JavaScript runtime and the DOM subsystem. The labels ①-⑥ in the figure show: ① the host's DOM being cloned at the start of the transaction, ② the host's call stack before a call that suspends the transaction, ③ the call stack after suspension, ④ the host's call stack when the transaction is about to resume; the speculative DOM has been updated with the requested changes, ⑤ the host's call stack just after resumption, ⑥ shows the transaction committing, which copies all speculative changes to the host's DOM and JavaScript heap. The thick lines on the call stacks denote transaction delimiters. Arrows show control transfer from the transaction to the iblock and back.

when the host aborts the guest after it detects the clickjacking attempt, the host's DOM will not contain any remnants of the guest's actions (such as event handlers registered by the guest). The host's JavaScript objects, such as `Editor`, are also unaffected. Speculatively executing guests therefore allows hosts to cleanly recover from attack attempts.

Iblocks offer hosts the option to postpone external operations. For example, a host may wish to defer all network requests from an untrusted advertisement until the end of the transaction. It can do so using an iblock that buffers these requests when they suspend, and thereafter resume the transaction; the buffered requests can be processed after the transaction has completed. Such postponement will not affect the guest if the buffered requests are asynchronous, *e.g.,* `XMLHttpRequest`.

Because a transaction may suspend several times, the iblock is structured as a loop, whose body executes each time the transaction suspends and once when the transaction completes. This way, the same policy checks apply whether the transaction suspended or completed.

3 Design of Transcript

We now describe the design of Transcript's mechanisms using Figure 3, which summarizes the workflow of a Transcript-enhanced host. The figure shows the operation of the Transcript runtime system at key points during the execution of the host, which has included an untrusted guest akin to the one in Figure 1 using a transaction.

When a transaction begins execution, Transcript first provides the transaction with its private copy of the host's DOM tree. It does so by cloning the current state of the

host's DOM, including any event handlers associated with the nodes of the DOM (① in Figure 3). When a guest references nodes in the host's DOM, Transcript redirects these references to the corresponding nodes in the transaction's private copy of the DOM.

Next, the Transcript runtime pushes a *transaction delimiter* on the JavaScript call stack. Transcript places the activation records of methods invoked within the transaction above this delimiter. It also records the locations of JavaScript objects accessed/modified within the transaction in read/write sets. If the transaction executes an external operation, the runtime suspends the transaction. To do so, it creates a transaction object and (a) initializes the object with the transaction's read/write sets; (b) pops all the activation records on the JavaScript call stack until the topmost transaction delimiter; (c) stores these activation records in the transaction object; (d) saves the program counter; and (e) sets the program counter to immediately after the end of the transaction, *i.e.,* the start of the iblock (steps ② and ③ in Figure 3).

The iblock logically extends from the end of the transaction to the last `resume` or `commit` call on the transaction object (*e.g.,* lines A–R in Figure 2). The iblock can access the transaction object and its read/write sets to make policy decisions. If the iblock invokes `resume` on a suspended transaction, the Transcript runtime (a) pushes a transaction delimiter on the current JavaScript call stack; (b) pushes the activation records saved in the transaction object; and (c) restores the program counter to its saved value. Execution therefore resumes from the statement following the external operation (see ④ and ⑤). If the iblock invokes `commit` instead, the Transcript runtime updates the JavaScript heap using the values in the transaction object's write set. The `commit` operation also replaces the host's DOM with the cloned DOM (step ⑥).

The Transcript runtime behaves in the same way even when transactions are nested: Transcript pushes a new delimiter on the JavaScript call stack for each level of nesting encountered at runtime. Each suspend operation only pops activation records until the topmost delimiter on the stack. Nesting is important when a guest itself wishes to confine code that it does not trust. This situation arises when a host includes a guest from a first-tier advertising agency (`1sttier.com`), which itself includes code from a second-tier agency (`2ndtier.com`). Whether the host confines the advertisement using an *outer* transaction, `1sttier.com` may itself confine code from `2ndtier.com` using an *inner* transaction using its own security policies. If code from `2ndtier.com` attempts to modify the DOM, that call suspends and traps to the iblock defined by `1sttier.com`. If this iblock attempts to modify the DOM on behalf of `2ndtier.com`, the outer transaction suspends in turn and passes control to the host's iblock. In effect, the DOM modification succeeds only if it is permitted at *each* level of nesting.

3.1 Components of an Iblock

As discussed in Section 2, an iblock consists of two parts: a *host-specific* part, which codifies the host's policies to mediate guests, and a *mandatory* part, which contains functionality that is generic to all hosts. In our implementation, we have encoded the second part as a JavaScript library (`libTranscript`) that can simply be included into the iblock of a host. This mandatory part implements two functionalities: gluing execution contexts and generating wrappers for event handlers.

Gluing Execution Contexts. Guests often use `document.write` or similar calls to modify the host's DOM, as shown on line 9 of Figure 1. When such guests execute within a transaction, the `document.write` call traps to the iblock, which must complete the call on behalf of the guest and render the HTML in the cloned DOM. However, the HTML code in `document.write` may contain scripts, *e.g.,* `document.write('<script src = code.js>')`. The execution of `code.js`, having been triggered by the guest, must then be mediated by the same security policy that governs the guest.

Thus, `code.js` should be executed in the same context as the transaction where the guest executes. To achieve this goal, the mandatory part of the iblock encapsulates the content of `code.js` into a function and uses a builtin `glueresume` method of the transaction object to instruct the Transcript runtime to invoke this function when it resumes the suspended transaction. The net effect is similar to fetching and inlining the content of `code.js` into the transaction. We call this operation *gluing*, because it glues the code in `code.js` to that of the guest.

To implement gluing, the iblock must recognize that the `document.write` includes additional scripts. This in turn requires the iblock to parse the HTML argument to `document.write`. We therefore exposed the browser's HTML parser through a new `document.parse` API to allow HTML (and CSS) parsing in iblocks. This API accepts a HTML string argument, such as the argument to `document.write`, and parses it to recognize `<script>` elements and other HTML content. It also recognizes inline event-handler registrations, so that they can be wrapped as described in Section 3.1. When the iblock invokes `document.parse` (in Figure 2, it is invoked within the call to `WriteToTxDOM` on line M), the parser creates new functions that contain code in `<script>` elements. It returns these functions to the host's iblock, which can then invoke them by gluing. The parser also renders other (non-script) HTML content in the cloned DOM.

Guest operations involving `innerHTML` are handled simlarly. Transcript suspends a guest that attempts an `innerHTML` operation, parses the new HTML code for any scripts, and glues their execution into the guest's context.

Generating Wrappers for Event Handlers. Guests executing within a transaction may attempt to register functions to handle asynchronous events. For example, line 8 in Figure 1 registers `displayAds` as an `onMouseOver` handler. Because `displayAds` is guest code, it is important to associate it with the iblock for the transaction that registered it and to subject it to the same policy checks. Transcript does so by creating a new function `tx_displayAds` that *wraps* `displayAds` within a transaction guarded by the same iblock, and registering `tx_displayAds` as the event handler for the `onMouseOver` event.

To this end, the mandatory part of the iblock includes creating wrappers (such as `tx_displayAds`) for event handlers. When the guest executes a statement such as `elem.addEventListener(...)`, it would trap to the iblock, which can then examine the arguments to this call and create a wrapper for the event handler. Guests can alternatively use `document.write` calls to register event handlers *e.g.,* `document.write ('<div onMouseOver="displayAds();">')`. In this case, the iblock recognizes that an event handler is being registered by parsing the HTML argument of the `document.write` call (using the `document.parse` API) when it suspends, and wraps the call. Our wrapper generator handles all the event models supported by Firefox [47].

Besides event handlers, JavaScript supports other constructs for asynchronous execution: AJAX callbacks, which execute upon receiving network events (`XMLHttpRequest`), and features such as `setTimeOut` and `setInterval` that trigger code execution based upon timer events. The mandatory part of the iblock also handles these constructs by wrapping callbacks as just described.

3.2 Hiding Sensitive Variables

The iblock of a transaction checks the guest's actions against the host's policies. These policies are themselves encoded in JavaScript, and may use methods and variables (*e.g.,* `tx`, `tocommit` and `builtins` in Figure 1) that must be protected from the guest. Without precautions, the guest can use JavaScript's extensive reflection capabilities to tamper with these sensitive variables. Figure 4 presents an example of one such attack, a reference leak, where the malicious guest obtains a reference to the `tx` object by enumerating the properties of the `this` object, and redefines the method `tx.getWriteSet` speculatively. As presented, the example in Figure 1 is vulnerable to such a reference leak.

To protect such sensitive variables, we adopt a defense called *variable hiding* that eliminates the possibility of leaks by construction. This technique mandates that guests be placed outside the scope of the iblock's variables, such as `tx`. The basic idea is to place the guest and the iblock in separate, lexically scoped functions, so that variables such as `tx`, `tocommit` and `builtins` are not accessible to the guest. By so hiding sensitive variables from the guest, this defense prevents reference leaks. Figure 8 illustrates this defense after introducing some more details of our implementation.

```
var tx = transaction { ... //code that suspends ...
    for (var x in this) {
        if (this[x] instanceof Tx_obj) txref = this[x];
    }; txref.getWriteSet = function() { };
}
```

Fig. 4. A guest that implements a reference leak. The `tx` object is created and attached to `this` when guest suspends.

4 Security Assurances

Transcript's ability to protect hosts from untrusted guests depends on two factors: (a) the assurance that a guest cannot subvert Transcript's mechanisms, *i.e.,* the robustness of the trusted computing base; and (b) host-specific policies used to mediate guests.

4.1 Trusted Computing Base

Transcript's trusted computing base (TCB) consists of the runtime component implemented in the browser and the mandatory part of the host's iblock. The TCB provides the following security properties: (a) *complete mediation*, *i.e.,* control over all JavaScript and external operations performed by a guest; and (b) *isolation*, *i.e.,* the ability to confine the effects of the guest.

(**1**) **Complete Mediation.** The Transcript runtime and the mandatory part of the host's iblock together ensure complete mediation of guest execution. The runtime: (a) records

all guest accesses to the host's JavaScript heap in the corresponding transaction's read/write sets; (b) causes a trap to the host's iblock when the guest attempts an external operation; and (c) redirects all guest references to the host's DOM to the cloned DOM. The mandatory part of the iblock, consisting of wrapper generators and the HTML parser, ensures that any additional code fetched by the guest or scheduled for later execution (*e.g.,* event handlers or callbacks for `XMLHttpRequest`) will itself be enclosed within transactions mediated by the same iblock. This process recurs so that the host's policies mediate all guest code, even event handlers installed by callbacks of event handlers.

(**2**) **Isolation.** Transcript isolates guest operations using speculative execution. It records changes to the host's JavaScript heap within the guest transaction's write set, and changes to the host's DOM within the cloned DOM. The host then has the opportunity to review these speculative changes within its iblock and ensure that they conform to its security policies. Observe that a suspended/completed transaction may provide the host with references to objects modified by the guest, *e.g.,* in Figure 1, a reference to `elem` is passed to the iblock via the `getObject` API. Speculative execution ensures that if the transaction has not yet been committed, then accesses to the object's methods and fields via this reference will still resolve to their values at the beginning of the transaction. Thus, for instance, a call to the `toString` method of the `elem` object in the iblock of Figure 1 would still work as intended if even if the guest had redefined this method within the transaction. Note that variables hidden from the guest cannot even be *speculatively* modified, thereby automatically isolating them from the guest.

Together, the above properties ensure the following invariant: At the point when a transaction suspends or completes execution and is awaiting inspection by the host's iblock, none of the host's JavaScript objects or its DOM would have been modified by the guest. Further, host variables hidden from the guest will not be modified even after the transaction has committed. Overall, executing a transaction never incurs any side effect, and any side effect that would be incurred by committing a transaction can be first vetted by inspecting the transaction.

4.2 Whitelisting for Host Policies

Hosts can import the speculative changes made by a guest after inspecting them against their security policies. Even though complete mediation and isolated execution ensure that the core *mechanisms* of Transcript cannot be subverted by guest execution (*i.e.,* they ensure that all of the guest's speculative actions will be available for inspection by the host), the ability of the host to isolate itself from the guest ultimately depends on its *policies*.

Host policies are necessarily domain-specific and have to be written manually in our current prototype. Though our experiments (Section 6.4) suggest that the effort required to write policies in Transcript is comparable to that required in other systems, writing policies is admittedly a difficult exercise and further research is needed to develop tools for policy authors to debug/verify the completeness of their policies. However, iblock policies once written can be reused across applications if applications share similar protection criteria. As a deployment model, we envision a vendor or community-driven

curated database of commonly-used iblock policies, which hosts can use to secure untrusted guests.

We suggest that iblock authors should employ a whitelist which specifies the host objects that can legitimately be modified by the guest and reject attempts to modify objects outside the whitelist. This guideline may cause false positives if the whitelist is not comprehensive. For example, both `window.location` and `window.location.href` can be used to change the location field of the host, but a whitelist that includes only one will reject guests that modify guest location using the other. Nevertheless, whitelisting allows hosts to be conservative when allowing guests to modify their objects.

5 Implementation in Firefox

We implemented Transcript by modifying Firefox (version 3.7a4pre). Overall, our prototype adds or modifies about 6,400 lines of code in the browser [1]. The bulk of this section describes Transcript's enhancements to SpiderMonkey (Firefox's JavaScript interpreter) (Section 5.1) and its support for speculative DOM updates (Section 5.2). We also discuss Transcript's support for conflict detection (Section 5.3) and the need to modify the `<script>` tag (Section 5.4).

5.1 Enhancements to SpiderMonkey

Our prototype enhances SpiderMonkey in five ways:

• *Transaction objects.* We added a new class of JavaScript objects to denote transactions. This object stores a pointer to the read/write sets, activation records of the transaction, and to the cloned DOM. It implements the builtin methods shown in Figure 5.
• *A* `transaction` *keyword.* We added a `transaction` keyword to the syntax of JavaScript. When the Transcript-enhanced JavaScript parser encounters this keyword, it (a) compiles the body of the transaction into an anonymous function; (b) inserts a new instruction, JSOP_BEGIN_TX, into the generated bytecode to signify the start of a transaction; and (c) inserts code to invoke the anonymous function. The transaction ends when the anonymous function completes execution. Finally, the anonymous function returns a transaction object when it suspends or completes execution.
• *Read/write sets.* Transcript adds read/write set-manipulation to the interpretation of several JavaScript bytecode instructions. We enhanced the interpreter so that each bytecode instruction that accesses or modifies JavaScript objects additionally checks whether its execution is within a transaction (*i.e.,* if an unfinished JSOP_BEGIN_TX was previously encountered in the bytecode stream). If so, the execution of the instruction also logs an identifier denoting the JavaScript object (or property) accessed/modified in its read/write sets, which we implemented using hash tables. We used SpiderMonkey's identifiers for JavaScript objects; references using aliases to the same object will return the same identifier.

[1] Transcript's design does not impose any fundamental restrictions on JITing of code within a transaction. However, to ease the development effort for our Transcript prototype, we chose not to handle JITed code paths in the prototype.

API	Description
`getReadSet`	Exports transaction's read set to JavaScript.
`getWriteSet`	Exports transaction's write set to JavaScript.
`getTxDocument`	Returns a reference to the speculative `document` object.
`isSuspended`	Returns `true` if the transaction is suspended.
`getCause`	Returns cause of a transaction suspend.
`getObject`	Returns object reference on which a suspension was invoked.
`getArgs`	Returns set of arguments involved in a transaction suspend.
`resume`	Resumes suspended transaction.
`glueresume`	Resumes suspended transaction and glues execution contexts.
`isDOMConflict`	Checks for conflicts between the host's and cloned DOM.
`isHeapConflict`	Checks for conflicts between the host and guest heaps.
`commit`	Commits changes to host's JavaScript heap and DOM.

Fig. 5. Key APIs defined on the transaction object

• *Suspend.* We modified the interpreter's implementation of bytecode instructions that perform external operations and register event handlers to suspend when executed within a transaction. The suspend operation and the builtin `resume` function of transaction objects are implemented as shown in Figure 3. We also introduced a `suspend` construct that allows hosts to customize transaction suspension. Hosts can include this construct within a transaction (before including guest code) to register custom suspension points. The call `suspend [obj.foo]` suspends the transaction when it invokes `foo` (if it is a method) or attempts to read from or write to the property `foo` of `obj`.
• *Garbage Collection.* We interfaced Transcript with the garbage collector to traverse and mark all heap objects that are reachable from live transaction objects. This avoids any inadvertent garbage collection of objects still reachable from suspended transactions that could be resumed in the future.

Integrating these changes into a legacy JavaScript engine proved to be a challenging exercise. We refer interested readers to Appendix A for a description of how our implementation addressed one such challenge, non-tail recursive calls in SpiderMonkey.

5.2 Supporting Speculative DOM Updates

Transcript provides each executing transaction with its private copy of the host's document structure and uses this copy to record all DOM changes made by guest code. This section presents notable details of the implementation of Transcript's DOM subsystem.

Transcript constructs a replica of the host's DOM when it encounters a JSOP_BEGIN_TX instruction in the bytecode stream. It clones nodes in the host's DOM tree, and iterates over each node in the host's DOM to copy references to any event handlers and dynamically-attached JavaScript properties associated with the node. If a guest attempts to modify an event handler associated with a node, the reference is rewritten to point to the function object in the transaction's write set.

Crom [35] also implemented DOM cloning for speculative execution (albeit not for the purpose of mediating untrusted code). Unlike Crom, which implemented DOM cloning as a JavaScript library, Transcript implements cloning in the browser itself. This feature simplifies several issues that Crom's designers faced (*e.g.,* cloning DOM-level 2 event handlers) and also allows efficient cloning.

When a guest references a DOM node within a transaction, Transcript transparently redirects this reference to the cloned DOM. It achieves this goal by modifying

the browser to tag each node in the host's DOM with a unique identifier (`uid`). During cloning, Transcript assigns each node in the cloned DOM the same `uid` as its counterpart in the host's DOM. When the guest attempts to access a DOM node, Transcript retrieves the `uid` of the node and walks the cloned DOM for a match. We defined a `getElementByUID` API on the `document` object to return a node with a given `uid`.

If the guest's operations conform to the host's policies, the host commits the transaction, upon which Transcript replaces the host's DOM with the transaction's copy of the DOM, thereby making the guest's speculative changes visible to the host.

5.3 Conflict Detection

When a host decides to commit a transaction, Transcript will replace the host's DOM with the guest's DOM. Objects on the host's heap are also overwritten using the write set of the guest's transaction. During replacement, care must be taken to ensure that the host's state is consistent with the guest's state. Consider, for instance, a guest that performs an `appendChild` operation on a DOM node (say node `N`). This operation causes a new node to be added to the cloned DOM, and also suspends the guest transaction. However, the host may delete node `N` before resuming the transaction; upon resumption, the guest continues to update a stale copy of the DOM (*i.e.,* the cloned version). When the transaction commits, the removed DOM node will be added to the host's DOM.

Transcript adds the `isDOMConflict` and `isHeapConflict` APIs to the transaction object, which allow host developers to register conflict detection policies. When invoked in the host's iblock, the `isDOMConflict` API invokes the conflict detection policy on each DOM node speculatively modified within the transaction (using the transaction's write set to identify nodes that were modified). The `isHeapConflict` API likewise checks that the state of the host's heap matches the state of the guest's heap at the start of the transaction. The snippet in Figure 6 shows one example of such a conflict detection policy (using `isDOMConflict`) encoded in the host's iblock that verifies that each node speculatively modified by the guest (`txNode`) has a parent in the host's DOM.

```
function hasParent(txNode) {
    var parent = txNode.parentNode;
    if (document.getElementByUID(parent.uid) != null) return true;
    else return false;
} ...
var isAllowed = tx.isDOMConflict(hasParent); // tx is the transaction object
```

Fig. 6. Example showing conflict detection

While Transcript provides the core *mechanisms* to detect transaction conflicts, it does not dictate any *policies* to resolve them. The host must resolve such conflicts within the application-specific part of its iblocks.

5.4 The <script> Tag

The examples presented thus far show hosts including guest code by inlining it within a transaction. However, hosts typically include guests using `<script>` tags, *e.g.,* `<script src="http://untrusted.com/guest.js">`. Transcript also supports code inclusion using `<script>` tags. To do so, it extends the `<script>` tag so that the fetched code can be encapsulated in a function rather than run immediately. The host application can use the modified `<script>`

tag as: `<script src="http://untrusted.com/guest.js" func="foobar">`. This tag encapsulates the code in `foobar`, which the host can then invoke within a transaction.

By itself, this modification unfortunately affects the scope chain in which the fetched code is executed. JavaScript code included using a `<script>` tag expects to be executed in the global scope of the host, but the modified `<script>` tag would put the fetched code in the scope of the function specified in the `func` attribute (*e.g.,* `foobar`).

We addressed this problem using a key property of `eval`. The ECMAScript standard [9, Section 10.4.2] specifies that an *indirect* `eval` (*i.e.,* via a reference to the `eval` function) is executed in the global scope. We therefore extracted the body of the compiled function `foobar` and executed it using an indirect `eval` call within a transaction (see Figure 8). This transformation allowed all variables and functions declared in the function `foobar` to be speculatively attached to the host's global scope.

6 Evaluation

We evaluated four aspects of Transcript. First, in Section 6.1 we study the applicability of Transcript to real-world guests, which varied in size from about 1,400 to 7,500 lines of code. Second, we show in Section 6.2 that a host that uses Transcript can protect itself and recover gracefully from malicious and buggy guests. Third, we report a performance evaluation of Transcript in Section 6.3. Last, in Section 6.4, we study the complexity of writing policies for Transcript. All experiments were performed with Firefox v3.7a4pre on a 2.33Ghz Intel Core2 Duo machine with 3GB RAM and running Ubuntu 7.10.

6.1 Case Studies on Guest Benchmarks

To evaluate Transcript's applicability to real-world guests, we experimented with five JavaScript applications, shown in Figure 7. For each guest benchmark in Figure 7, we played the role of a host developer attempting to include the guest into the host, *i.e.,* we created a Web page and included the code of the guest into the page using `<script>` tags. Most of the guests were implemented in several files; the `<script>` column in Figure 7 shows the number of `<script>` tags that we had to use to include the guest into the host. We briefly describe these guest benchmarks and the domain-specific policies that were implemented for each iblock.

	Benchmark	Size (LoC)	`<script>` tags
1	JavaScript Menu [7]	1,417	1
2	Picture Puzzle [40]	1,709	3
3	GoogieSpell [38]	2,671	4
4	GreyBox [39]	2,338	7
5	Color Picker [6]	7,543	6

Fig. 7. Guest benchmarks. We used transactions to isolate each of these benchmarks from a simple hosting Web page

(1) JavaScript Menu is a standalone widget that implements pull-down menus. Figure 8 shows how we confined JavaScript Menu using Transcript. The iblock for JavaScript menu enforced a policy that disallowed the guest from accessing the network (`XMLHttpRequest`) or domain cookies.

JavaScript Menu makes extensive use of `document.write` to build menus, with several of these calls used to register event handlers, as shown below (event handler registrations are shown in bold). Each `document.write` call causes the transaction to suspend and pass control to the iblock. The iblock uses `document.parse` to (a) parse the arguments to identify

```
1  <script src="jsMenu.js" func="menu"></script>
2  <script src="libTranscript.js></script>
3  <script>(function () {
4      var to_commit = true, e = eval;     // indirect eval
5      var tx = transaction { e(getFunctionBody(menu));}
6      to_commit = gotoIblock(tx);
7      if(to_commit) tx.commit();
8  })(); </script>
```

Fig. 8. Confining JavaScript Menu. (a) lines 1 and 5 demonstrate the enhanced `<script>` tag and the host's use of indirect `eval` to include the guest, which is compiled into a function (called `menu`; line 1) (Section 5.4). `getFunctionBody` extracts the code of the function `menu`; (b) line 3 implements variable hiding (Section 3.2), making `tx` invisible to the guest; (c) our supporting library `libTranscript` (line 2) implements the mandatory part of the iblock and is invoked from `gotoIblock`.

the HTML element(s) being created; (b) identify whether any event handlers are being registered and wrap them; and (c) write resulting HTML to the transaction's speculative DOM.

(2) Picture Puzzle uses the drag-and-drop features provided by the AJS JavaScript library [2] to build an application that prompts the user to arrange jumbled pieces of a picture within a 3×3 grid (we adapted this benchmark from [40]). We ran the benchmark within a transaction and enforced a domain-specific security policy that prevented the transaction from committing its changes if it attempted to install a handler to capture the user's keystrokes (*e.g.,* any event with `onkey` as a substring).

(3) GoogieSpell extends the AJS library to provide a spell-checking service. When a user clicks the "check spelling" button, GoogieSpell sends an `XMLHttpRequest` to a third-party server to fetch suggestions for misspelled words. We created a transactional version of GoogieSpell, whose iblock implemented a domain-specific policy that prevents an `XMLHttpRequest` once the benchmark has read domain cookies or if the target URL of `XMLHttpRequest` does not appear on a whitelist.[2]

(4) GreyBox is content-display application that also extends the AJS library. It can be used to display external pages, build image galleries, receive file uploads and even show video or Flash content. The application creates an `<iframe>` to load new content. Our transactional version of the GreyBox application encoded a domain-specific iblock policy that only allowed the creation of `<iframe>`s to whitelisted URLs.

(5) Color Picker builds upon the popular jQuery library [5] and lets a user pick a color by moving sliders depicting the intensities of red, blue and green. We executed the entire benchmark (including all the supporting jQuery libraries) as a transaction and encoded an iblock that disallowed modifications to the `innerHTML` property of arbitrary `<div>` nodes.

However, for this guest, it turns out that an iblock that disallows any changes to the sensitive `innerHTML` property of *any* `<div>` element is overly restrictive. This is because Color Picker modified the `innerHTML` property of a `<div>` element that it created. We therefore loosened our policy into a history-based policy that let the benchmark change `innerHTML` properties of `<div>` elements that it created. The iblock determines whether a `<div>` element was created by the transaction by querying its write set. The relevant snippet from the iblock is shown below; the `tx` variable denotes the transaction:

[2] Such *cross-origin resource sharing* permits cross-site `XMLHttpRequest`s, and is supported by Firefox-3.5 and higher [37].

```
1  var ws = tx.getWriteSet(); ...
2  if (tx.getCause().match("innerHTML") && ws.checkMembership(tx.getObject(), "*")
     && !(tx.getObject() instanceof HTMLBodyElement))
3      // perform action on behalf of untrusted code
```

6.2 Fault Injection and Recovery

To evaluate how Transcript can help hosts detect and debug malicious guest activity, we performed a set of fault-injection experiments on a real Web application that allows integration of untrusted guest code. We used the Bigace Web content management system [3] running on our Web server as the host, and created a Web site that mashed content from Bigace with content provided by untrusted guests (each guest was included into the mashup using the `<script>` tag). We wrote guests that emulated known attacks and studied host behavior when the host (1) directly included the guest in its protection domain; and (2) used Transcript to isolate the guest.

Our experiments show that with appropriate iblock policies, speculative execution ensured clean recovery; neither the JavaScript heap nor the DOM of the host was affected by the misbehaving guest.

(**1**) *Misplaced Event Handler.* JavaScript provides a `preventDefault` method that suppresses the default action normally taken by the browser as a result of the event. For example, the default action on clicking a link is to fetch the page corresponding to the URL referenced in the link. Several sites use `preventDefault` to encode domain-specific actions instead, *e.g.,* displaying a popup when a link is clicked.

In this experiment, we created a buggy guest that displays an advertisement within a `<div>` element. This guest mistakenly registers an `onClick` event handler that uses `preventDefault` with the `document` object instead of with the `<div>` element. The result of including this guest directly into the host's protection domain is that all hyperlinks on the Web page are rendered unresponsive. We then modified the host to isolate the guest using a policy that disallows a transaction to commit if it attempts to register an `onClick` handler with the `document` object. This prevented the advertisement from being displayed, *i.e.,* the `<div>` element containing the misbehaving guest was not even created, but otherwise allowed the host to function correctly. JavaScript reference monitors proposed in prior work can prevent the registration of the `onClick` handler, but leave the `div` element of the misbehaving guest on the host's Web page.

(**2**) *Prototype Hijacking.* We implemented a prototype hijacking attack by writing a guest that set the `Array.prototype.slice` function to `null`. To illustrate the ill-effects of this attack, we modified the host to include two popular (and benign) widgets, namely Twitter [8] and AddThis [1], in addition to the malicious guest. The prototype hijacking attack prevented both the benign widgets from functioning properly.

However, when the malicious guest is enclosed within a transaction whose iblock prevents a commit if it detects prototype hijacking attacks, the host and both benign widgets worked normally. We further inspected the transaction's write set and verified that none of the heap operations attributed to the malicious guest were actually applied to the host. Although traditional JavaScript reference monitors can detect and prevent prototype hijacking attacks by blocking further `<script>` execution, they do not allow the hosts to cleanly recover from all heap changes.

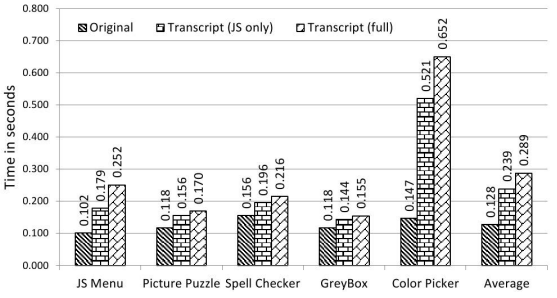


Fig. 9. Performance of guest benchmarks. This chart compares the time to load the unmodified version of each guest benchmark against the time to load the transactional version in the two variants of Transcript.

(**3**) *Oversized Advertisement.* We created a guest that displayed an interactive JavaScript advertisement within a `<div>` element. In an unprotected host, this advertisement expands to occupy the full screen on a `mouseover` event, *i.e.,* the guest registered a misbehaving event-handler that modifies the size of the `<div>`. We modified the host to isolate this guest using a transaction and an iblock that prevents a `commit` if the size of the `<div>` element increased beyond a pre-specified limit. With this policy, we observed that the host could successfully prevent the undesired `<div>` modification by discarding the speculative DOM and JavaScript heap changes made by the event handler executing within the transaction.

6.3 Performance

We measured the overhead imposed by Transcript both using guest benchmarks, to estimate the overall cost of using transactions, and with microbenchmarks, to understand the impact on specific JavaScript operations.

Guest Benchmarks. To evaluate the overall performance impact of Transcript, we measured the increase in the load time of each guest benchmark. Recall that each benchmark is included in the Web page using a set of `<script>` tags; the version that uses Transcript executes the corresponding JavaScript code within a single transaction using modified `<script>` tags. The `onload` event fires at the end of the document loading process, *i.e.,* when all scripts have completed execution. We therefore measured the time elapsed from the moment the page is loaded in the browser to the firing of the `onload` event.

To separately assess the impact of speculatively executing JavaScript and DOM operations, each experiment involved executing the benchmarks on two separate variants of Transcript, namely Transcript (full) which supports both speculative DOM and JavaScript operations and Transcript (JS only) which only supports speculative JavaScript operations (and therefore does not isolate DOM operations of the guest). Figure 9 presents the results averaged over 25 runs of this experiment. On average, Transcript (JS only) increased load time by just 0.11 seconds while Transcript (full) increased the load time by 0.16 seconds. These overheads are typically imperceptible to end users. Only Color Picker had above-average overheads. This was because (a) the

guest heavily interacted with the DOM, causing frequent suspension of its transaction; and (b) the guest had several `Array` operations that referenced the `length` of the array. Each such operation triggered a traversal of read/write sets to calculate the array length.

Note that Transcript only degrades performance of JavaScript code executing within transactions (*i.e.,* guests). The performance of code executing outside transactions (*i.e.,* hosts) is not affected by our prototype.

Microbenchmarks. We further dissected the performance of Transcript using microbenchmarks designed to stress specific functionalities. We used two sets of microbenchmarks: function calls and event dispatchers. In our experiments, we executed each microbenchmark within a transaction whose iblock simply permitted all actions and resumed the transaction without enforcing additional security policies, and compared its performance against the non-transactional version.

Microbenchmark	Overhead
Native Functions	
`eval("1")`	6.69×
`eval("if (true)true;false")`	6.87×
`fn.call(this, i)`	1.89×
External operations	
`getElementById("checkbox")`	6.78×
`getElementsByTagName("input")`	6.89×
`createElement("div")`	3.69×
`createEvent("MouseEvents")`	3.82×
`addEventListener("click", clk, false)`	26.51×
`dispatchEvent(evt)`	1.20×
`document.write("<b>Hi</b>")`	1.26×
`document.write("<script>x=1;</script>")`	2.01×

Fig. 10. Performance of function call microbenchmarks

Function calls. We devised a set of microbenchmarks (Figure 10) that stress the performance of Transcript's function call-handling code. Each benchmark invoked the code in first column of Figure 10 10,000 times.

Recall that Transcript suspends on function calls that cause external operations and for certain native function calls, such as `eval`. Each suspend operation requires Transcript to save the state of the transaction, execute the iblock, and restore the transaction state upon the execution of a `resume` call. Most of the benchmarks in Figure 10 trigger a suspension, which induces significant overheads. In particular, `addEventListener` had an overhead of 26.51×. The bulk of the overhead was induced by code in the iblock that generates wrappers for the event handler registered using `addEventListener`.

User Events. A JavaScript application executing within a transaction may dispatch user events, such as mouse clicks and key presses, which must be processed by the event handler associated with the relevant DOM node. The promptness with which events are dispatched typically affects end-user experience.

Event name	Overhead	
	Normalized	Raw (μs)
Drag Event (`drag`)	1.71×	97
Keyboard Event (`keypress`)	1.16×	150
Message Event (`message`)	1.17×	85
Mouse Event (`click`)	1.54×	86
Mouse Event (`mouseover`)	2.05×	88
Mutation Event (`DOMAttrModified`)	2.14×	88
UI Event (`overflow`)	1.97×	61

Fig. 11. Performance of event dispatch microbenchmarks

To measure the impact of transactions on this aspect of browser performance, we devised a set of microbenchmarks that dispatched user events such as clicking a checkbox, moving the mouse, pressing keys, etc. and measured the delay in handling them (Figure 11).

In each case, code that generated and dispatched the event executed as a transaction with an iblock that allowed all actions. To measure overhead, we executed this code 1,000 times and compared its performance against a native event dispatcher. Figure 11

Policy	T-LOC	C-LOC	Policy	T-LOC	C-LOC
Conscript-#1	7	2	Conscript-#2	5	6
Conscript-#3	6	3	Conscript-#4	9	7
Conscript-#5	9	9	Conscript-#6	5	8
Conscript-#7	7	5	Conscript-#8	5	6
Conscript-#10	9	16	Conscript-#11	12	17
Conscript-#12	5	4	Conscript-#13	4	6
Conscript-#14	3	5	Conscript-#15	6	7
Conscript-#16	6	4	Conscript-#17	7	5

Fig. 12. Policy complexity. Comparing policies in Transcript (T-LOC) and Conscript (C-LOC). Policies are numbered as in Conscript [34]. We omitted Conscript-#9 since it is IE-specific.

presents the results, which show the normalized overhead as well as the raw delay to process a single event. As this figure shows, although the normalized overheads range from 16% to 114%, the raw delays average about 94 microseconds, which is imperceptible to end users.

6.4 Complexity of Policies

To study the complexity of writing policies in Transcript, we compared the number of lines of code needed to write policies in Transcript and in Conscript [34]. We considered the policies discussed in Conscript and wrote equivalent policies in Transcript; Figure 12 compares the source lines of code (counting number of semi-colons) of policies in Transcript and Conscript. This shows that the programming effort required to encode policies in both systems is comparable.

7 Related Work

This paper builds upon the idea of extending JavaScript with transactions, which was proposed in a recent position paper [14]. While that paper focused on the semantics of the extended language, this paper is the first to report the design and implementation of a complete speculative execution system for JavaScript.

There is much prior work in the broad area of isolating untrusted guests. Transcript is unique because it allows hosts to recover cleanly and easily from the effects of malicious or buggy guests (Figure 13). In exchange for requiring no modification to the guest, Transcript requires modifications both to the host (*i.e.,* the server side) and to the browser (*i.e.,* the client side) to enhance the JavaScript language.

Static Analysis. Despite the dynamic nature of JavaScript, there have been a few efforts at statically analyzing JavaScript code. Gatekeeper [21] presents a static analysis to validate widgets written in a subset of JavaScript. It does so by matching widget source code against a database of patterns denoting unsafe programming practices. Guha *et al.* [22] developed static techniques to improve AJAX security. Their work uses static analysis to enhance a server-side proxy with models of AJAX computation on the client. The proxy then ensures that AJAX requests from the client conform to these models.

Chugh *et al.* [12] developed a staged information-flow tracking framework for JavaScript to protect hosts from untrusted guests. Its static analysis identifies constraints on host variables that can be read or written by guests. It validates these constraints on

System	Recovery	Unrestricted guest	Unmodified browser	Policy coverage
Transcript	✓	✓	✗	Heap + DOM
Conscript [34]	✗	✓	✗	Heap + DOM
AdJail [26]	✗	✓	✓	DOM(1)
Caja [36]	✗	✗	✓	Heap + DOM
Wrappers [29,30,33]	✗	✓(2)	✓	Heap + DOM
Info. flow [12]	✗	✓	✓	Heap
IRMs [42,48,43]	✗	✓	✓	Heap + DOM
Subsetting [30,13,18]	✗	✗	✓	Static policies(3)

Fig. 13. Techniques to confine untrusted guests. (1) Adjail uses a separate `<iframe>` to disallows guests from executing in the host's context. (2) Some wrapper-based solutions [29] restrict JavaScript constructs allowed in guests. (3) Subsetting is a static technique and its policies are not enforced at runtime.

code loaded at runtime via `eval` or `<script>` tags, and rejects such code if it violates these constraints. Unlike Transcript, which tracks changes to both the heap and DOM, Chugh *et al.*'s work only tracks changes to the heap.

Language Restriction. Several projects have defined subsets of JavaScript that omit dynamic constructs, such as `eval`, `with` and `this`, to make it amenable to static analysis [13,18,36,21]. However, designing safe subsets of JavaScript is non-trivial [31,28,30,19], and also prevents code developers from using arbitrary constructs of the language in their applications. Transcript places no such restrictions on guest code.

Object Capabilities, Wrappers, and Code Rewriting. Object capability and wrapper-based solutions (*e.g.,* [33,30,29]) create wrapped versions of JavaScript objects to be protected, and ensure that such objects can only be accessed by code that has the capability to do so. In contrast to these techniques, which provide isolation by wrapping the host's objects, Transcript wraps guest code using transactions, and mediates its actions with the host via iblocks. Prior research has also developed solutions to inline runtime checks into untrusted guests. These include BrowserShield [43], CoreScript [48], and the work of Phung *et al.* [42]. Unlike these works, Transcript simply wraps untrusted code in a transaction, and does not modify it. These works also do not explicitly address recovery.

Aspect-Oriented Policy Enforcement. Aspect-oriented programming (AOP) techniques have previously been used to enforce cross-cutting security policies [17,10,16]. Among the AOP-based frameworks for JavaScript [34,23], our work is most closely related to Conscript [34], which uses runtime aspect-weaving to enforce policies on untrusted guests. Both Conscript and Transcript require changes to the browser to support their policy enforcement mechanisms. However, unlike Transcript, Conscript does not address recovery from malicious guests, and also requires guests to be written in a subset of JavaScript. While recovery may also be possible in hosts that use Conscript, the hosts would have to encode these recovery policies explicitly. In contrast, hosts that use Transcript can simply discard the speculative changes made by a policy-violating guest.

Browser-Based Sandboxing. Both BEEP [25] and MashupOS [45] enhance the browser with new HTML constructs. BEEP's constructs allow the browser to detect script-injection attacks, while MashupOS provides sandboxing constructs to improve the security of client-side mashups. While Transcript requires modified `<script>` tags as well,

it provides the ability to speculatively execute and observe the actions of untrusted code, which neither BEEP nor MashupOS provide.

AdJail aims to protect hosts from malicious advertisements [26]. It confines advertisements by executing them in a separate `<iframe>`, and uses `postMessage` to allow the `<iframe>` to communicate with the host. Hosts use access control policies to determine the set of DOM modifications allowed by an advertisement. AdJail is effective at confining advertisements, which cannot affect the host's heap. However, it is unclear whether this approach will work in scenarios where hosts and guests need to interact extensively, *e.g.,* in the case where the guest is a library that the host wishes to use. The forthcoming EcmaScript 6 / Harmony modules [15] and HTML5 `<iframe sandbox>` attribute [24] also enable new isolation mechanisms by constraining the way guest code interacts with the host, but unlike Transcript they do not address recovery.

Sandboxing through Speculation. Blueprint [27] and Virtual Browser [11] confine guests by setting up a virtual environment for their execution. This environment is itself written in JavaScript and parses HTML and script content, thereby mediating the execution of guests on unmodified browsers. However, unlike Transcript, they do not address recovery. Transcript is most closely related to Worlds [46] in its motivation to provide first-class primitives that enable programmers to contain side-effects. However, there are major design and implementation differences including Transcript's ability to enforce fine-grained security policies and its implementation in SpiderMonkey.

Using Transactions for Performance. Crom [35] applies speculation to event handlers and takes non-speculative event handlers to create speculative versions, running them in a cloned browser context. ParaScript [32] implements a selective checkpointing scheme which avoids JavaScript constructs that allow code injection like document.write, innerHTML, etc., and stops speculation if checkpointing becomes expensive. Both, Crom and ParaScript use speculation to improve performance. In contrast, Transcript addresses all scenarios in the design and implementation of a fully speculative JavaScript engine and required several new contributions, such as the ability to suspend/resume transactions and wrap event handlers.

8 Conclusion

Our research shows that extending JavaScript with support for transactions allows hosting Web applications to speculatively execute and enforce security policies on untrusted guests. Speculative execution allows hosts to cleanly and easily recover from the effects of malicious and misbehaving guests. In building Transcript, we made several contributions, including suspend/resume for JavaScript, support for speculative DOM updates, and novel strategies to implement transactions in commodity JavaScript interpreters.

Acknowledgements. We thank James Mickens and the anonymous reviewers for their comments. This work was supported by NSF award 0952128.

References

1. Addthis, http://www.addthis.com/
2. AJS: The ultra lightweight JavaScript library, http://orangoo.com/labs/AJS/
3. BIGACE web content management system, http://www.bigace.de/
4. Dom-based xss injection, https://www.owasp.org/index.php/Interpreter_Injection#DOM-based_XSS_Injection
5. jQuery: The write less, do more, JavaScript library, http://jquery.com
6. Jquery UI slider plugin, http://jqueryui.com/demos/slider
7. JavaScript widgets/menu, http://jswidgets.sourceforge.net
8. Twitter/profile widget, http://twitter.com/about/resources/widgets/widget_profile
9. ECMAScript language spec., ECMA-262, 5th edn. (December 2009)
10. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with Polymer. In: ACM PLDI (2005)
11. Cao, Y., Li, Z., Rastogi, V., Chen, Y.: Virtual browser: a Web-level sandbox to secure third-party JavaScript without sacrificing functionality (poster). In: ACM CCS (2010)
12. Chugh, R., Meister, J., Jhala, R., Lerner, S.: Staged information flow in JavaScript. In: ACM SIGPLAN PLDI (2009)
13. Crockford, D.: ADsafe - Making JavaScript safe for advertising, http://adsafe.org
14. Dhawan, M., Shan, C.-C., Ganapathy, V.: Position paper: The case for JavaScript transactions. In: 5th ACM SIGPLAN PLAS Workshop (June 2010)
15. ECMAScript. Harmony modules, http://wiki.ecmascript.org/doku.php?id=harmony:modules
16. Erlingsson, Ú.: The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Cornell University (2004)
17. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: IEEE S&P (1999)
18. Facebook. FBJS - Facebook developerwiki (2007)
19. Finifter, M., Weinberger, J., Barth, A.: Preventing capability leaks in secure JavaScript subsets. In: NDSS (2010)
20. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. 40 (June 2005)
21. Guarnieri, S., Livshits, B.: GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In: USENIX Security (2009)
22. Guha, A., Krishnamurthi, S., Jim, T.: Using static analysis for Ajax intrusion detection. In: WWW (2009)
23. Washizaki, H., et al.: AOJS: Aspect-oriented JavaScript programming framework for Web development. In: Intl. Wkshp. Aspects, Components, and Patterns for Infrastructure Software (2009)
24. Hickson, I.: Html iframe sandbox attribute, http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox
25. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: WWW (2007)
26. Louw, M.T., Ganesh, K.T., Venkatakrishnan, V.N.: Adjail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In: USENIX Security (2010)
27. Ter Louw, M., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: IEEE S&P (2009)

28. Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
29. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating JavaScript with Filters, Rewriting, and Wrappers. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 505–522. Springer, Heidelberg (2009)
30. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted Web applications. In: IEEE S&P (2010)
31. Maffeis, S., Taly, A.: Language based isolation of untrusted JavaScript. In: IEEE CSF (2009)
32. Mehrara, M., Hsu, P.-C., Samadi, M., Mahlke, S.: Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism. In: International Symposium on High-Performance Computer Architecture, pp. 87–98 (2011)
33. Meyerovich, L., Porter Felt, A., Miller, M.S.: Object views: Fine-grained sharing in browsers. In: WWW (2010)
34. Meyerovich, L., Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In: IEEE S&P (2010)
35. Mickens, J., Elson, J., Howell, J., Lorch, J.: Crom: Faster Web browsing using speculative execution. In: NSDI (2010)
36. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized JavaScript (2008) (manuscript)
37. Mozilla Developer Center. HTTP access control, http://developer.mozilla.org/En/HTTP_access_control
38. Orangoo-Labs. GoogieSpell, http://orangoo.com/labs/GoogieSpell
39. Orangoo-Labs GreyBox, http://orangoo.com/labs/GreyBox
40. Orangoo-Labs. Sortable list widget, http://orangoo.com/AJS/examples/sortable_list.html
41. Di Paola, S., Fedon, G.: Subverting Ajax: Next generation vulnerabilities in 2.0 Web applications. In: 23rd Chaos Communication Congress (2006)
42. Phung, P., Sands, D., Chudnov, A.: Lightweight self-protecting JavaScript. In: ASIACCS (2009)
43. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: Browsershield: Vulnerability-driven filtering of dynamic HTML. ACM Trans. Web 1(3), 11 (2007)
44. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. SIGSOFT Softw. Eng. Notes 30 (September 2005)
45. Wang, H.J., Fan, X., Howell, J., Jackson, C.: Protection and communication abstractions for web browsers in MashupOS. In: ACM SOSP (2007)
46. Warth, A., Ohshima, Y., Kaehler, T., Kay, A.: Worlds: Controlling the Scope of Side Effects. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 179–203. Springer, Heidelberg (2011)
47. WWW-Consortium. Document object model events (November 2000), http://www.w3.org/TR/DOM-Level-2-Events/events.html
48. Yu, D., Chander, A., Islam, N., Serikov, I.: JavaScript instrumentation for browser security. In: ACM POPL (2007)

A Non-tail-Recursive Interpreters

A key challenge in enhancing a legacy JavaScript interpreter, such as SpiderMonkey, with support for transactions is in how the interpreter uses recursion. To support the suspend/resume mechanism for switching control flow between a transaction and its iblock, the interpreter must not accumulate any activation records in its native stack (*e.g.,* the C++ stack, for SpiderMonkey) between when a transaction starts and when it suspends. In particular, the interpreter must not represent JavaScript function calls by C++ function calls. The same issue also arises when a compiler or JIT interpreter is used to turn JavaScript code into machine code.

To illustrate this point, consider SpiderMonkey, which implements the bytecode interpreter in C++. The main entry point to the bytecode interpreter is the C++ function JS_interpret, which maintains the JavaScript stack as a linked list of activation records, each of which is a C++ structure. When one function calls another in JavaScript, the JS_interpret function does not call itself in C++; instead, it adds a new activation record to the front of the linked list and continues with the same bytecode interpreter loop as before. Similarly, when a function returns to another in JavaScript, JS_interpret does not return in C++; instead, it removes an old activation record from the front of the linked list and continues with the same bytecode interpreter loop as before. For the most part, SpiderMonkey does not represent JavaScript calls by C++ calls.

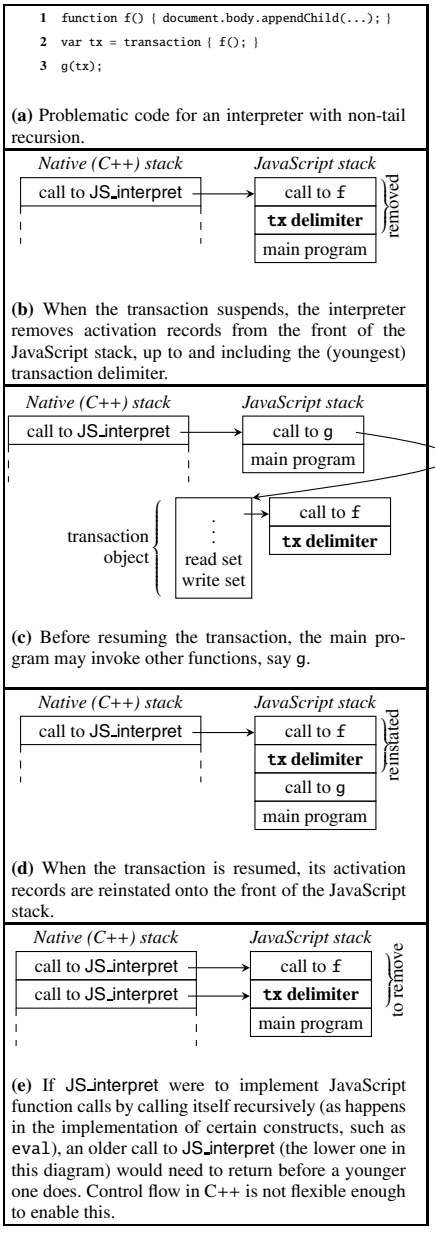


Fig. 14. Native versus JavaScript call stacks

The fact that SpiderMonkey does not represent JavaScript calls by native calls helps us add transactions to it without making invasive changes, as the following example illustrates. Suppose a transaction invokes a function f that suspends for some reason,

e.g., in Figure 14(a), the function `f` calls `appendChild`. If the C++ call to JS_interpret that executes the transaction were not same as the one that executes the called function `f`, then the former, although older, would have to return before the latter returns. As detailed in Figure 14, the former has to return when suspending the transaction, whereas the latter has to return when resuming the transaction. Even exception handling in C++ does not allow such control flow.

Unfortunately, JS_interpret in SpiderMonkey does call itself in a few situations. For example, it handles the `eval` construct in this way, and the problem of the C++ stack in Figure 14(e) does arise if we replace the `document.body.appendChild(...)` of Figure 14(a) by `eval("document.body.appendChild(...)")`. One way to solve this problem requires applying the continuation-passing-style transformation to the interpreter to put it into tail form, *i.e.,* convert all recursive calls to JS_interpret to tail calls. However, this transformation is invasive, especially if done manually on legacy interpreters.

Transcript uses a less invasive mechanism to enable suspend/resume in SpiderMonkey. This mechanism is similar in functionality to gluing (see Section 3.1), and we explain it with an example. Consider the `eval` construct, whose functionality is to parse its input string, compile it into bytecode, and then execute the bytecode as usual. Because only the last step, *i.e.,* that of executing the bytecode, can suspend, we simply changed the behavior of `eval` so that, if invoked inside a transaction, it suspends the transaction right away. The iblock of the transaction can then compile the string into bytecode and include the bytecode into the execution of the transaction. This is achieved by adding a new activation record to the front of the transaction's JavaScript stack and modifying the program counter to execute this code when the transaction resumes. When the suspended transaction resumes, it transfers control to the `eval`ed code, which can freely suspend. Besides `eval`, our current Transcript prototype also implements gluing for `document.write` (as discussed in Section 3.1) and JavaScript builtins `call` and `apply`, which make non-tail recursive calls to JS_interpret.

JavaScript as an Embedded DSL

Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky

Ecole Polytechnique Fédérale de Lausanne (EPFL)
first.last@epfl.ch

Abstract. Developing rich web applications requires mastering different environments on the client and server sides. While there is considerable choice on the server-side, the client-side is tied to JavaScript, which poses substantial software engineering challenges, such as moving or sharing pieces of code between the environments. We embed JavaScript as a DSL in Scala, using Lightweight Modular Staging. DSL code can be compiled to JavaScript or executed as part of the server application. We use features of the host language to make client-side programming safer and more convenient. We use gradual typing to interface typed DSL programs with existing JavaScript APIs. We exploit a selective CPS transform already available in the host language to provide a compelling abstraction over asynchronous callback-driven programming in our DSL.

Keywords: JavaScript, Scala, DSL, programming languages.

1 Introduction

Developing rich web applications requires mastering a heterogeneous environment: though the server-side can be implemented in any language, on the client-side, the choice is limited to JavaScript. The trend towards alternative approaches to client-side programming (as embodied by CoffeeScript [9], Dart [14] & GWT [17]) shows the need for more options on the client-side. How do we bring advances in programming languages to client-side programming?

One challenge in developing a large code base in JavaScript is the lack of static typing, as types are helpful for maintenance, refactoring, and reasoning about correctness. Furthermore, there is a need for more abstraction and modularity. "Inversion of control" in asynchronous callback-driven programming leads to code with control structures that are difficult to reason about. Another challenge is to introduce helpful abstractions without a big hit on performance and/or code size. Communication between the server side and the client side aggravates the impedance mismatch: in particular, data validation logic needs to be duplicated on the client-side for interactivity and on the server-side for security.

There are three widely known approaches for addressing the challenges outlined above. One is to create a standalone language or DSL that is compiled to JavaScript and provides different abstractions compared to JavaScript. Examples include WebDSL [36], Links [10,11] and Dart [14]. However, this approach usually requires a lot of effort in terms of language and compiler design, and

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 409–434, 2012.

tooling support, although WebDSL leverages Spoofax [19] to alleviate this effort. Furthermore, it is not always clear how these languages interact with other languages on the server-side or with the existing JavaScript ecosystem on the client-side.

Another approach is to start with an existing language like Java, Scala or Clojure and compile it to JavaScript. Examples include GWT [17], Scala+GWT [30] and Clojurescript [8]. This approach addresses the problem of impedance mismatch between client and server programming but comes with its own set of challenges. In particular, compiling Scala code to JavaScript requires compiling Scala's standard library to JavaScript as any non-trivial Scala program uses Scala collections. This leads to not taking full advantage of libraries and abstractions provided by the target platform which results in big code size and suboptimal performance of Scala applications compiled to JavaScript. For example, a map keyed by strings would be implemented natively in JavaScript as an object literal, while, in Scala, one would likely use the hash map from the standard library, causing it to be compiled to and emulated in JavaScript. Moreover, both approaches tend to not accommodate very well to different API design and programming styles seen in many existing JavaScript libraries. A drawback of this second approach is that the whole starting language needs to be translated to JavaScript: there is no easy or modular way to limit the scope of the source language. The F# Web Tools [26] are an interesting variation of this approach that employ F# quotations to translate only parts of a program and allow programmers to define custom mappings for individual data types.

A third approach is to design a language that is a thin layer on top of JavaScript but provides some new features. A prime example of this idea is CoffeeScript [9]. This approach makes it easy to integrate with existing JavaScript libraries but does not solve the impedance mismatch problem. In addition, it typically does not give rise to new abstractions addressing problems seen in callback-driven programming style, though some JavaScript libraries such as Flapjax [22] and Arrowlets [20] are specifically designed for this purpose.

We present a different approach, based on Lightweight Modular Staging (LMS) [28], that aims to incorporate good ideas from all the approaches presented above but at the same time tries to avoid their described shortcomings. LMS is a technique for embedding DSLs as libraries into a host language such as Scala, while enabling domain-specific compilation / code-generation. The program is split into two stages: the first stage is a program generator that, when run, produces the second stage program. Whether an expression belongs to the first or second stage is decided by its type. Expressions belonging to the second stage, also called "staged expressions", have type `Rep[T]` in the first stage when yielding a computation of type `T` in the second stage. Expressions evaluated in the first stage become constants at the second stage. Other approaches to staging include MetaML [34], LISP quasiquotations, and binding-time analysis in partial evaluation. Previous work has established LMS as a pragmatic approach to runtime code generation and compiled DSLs. In particular, the Delite framework [4,29,7] uses this approach to provide an extensible suite of

high-performance DSLs targeting heterogeneous parallel platforms (with options to generate code to Scala, C and Cuda) [21], for domains such as machine learning [33], numeric array processing [35] and mesh-based partial differential equation solvers [6]. LMS has also been used to generate SQL queries [37].

We propose to embed JavaScript as a DSL in a host language.[1] Through LMS (reviewed in section 2), we tackle the challenges outlined above with minimal effort, as most of the work is off-loaded to the host language. In particular, we make the following contributions:

- Our DSL is statically typed through the host language, yet supports gradual typing notably for incorporating external JavaScript libraries and APIs (section 3).
- In addition to generating JavaScript code, our DSL can be executed directly in the host language, allowing code to be shared between client and server (section 4).
- We use advanced object-oriented techniques to achieve modularity in our DSL: each language primitive and API is defined in a separate module (section 5).
- Our DSL supports typed object literals and class-based objects. The translations to JavaScript are lightweight and intuitive: the object literals translate to JSON-like object literals and the class-based objects to JavaScript constructor-based objects (section 6).
- On top of the straightforward embedding, we implement advanced abstractions in the host language. With minimal effort, we exploit the selective CPS transform already existing in Scala to provide a compelling abstraction over asynchronous callback-driven programming in our DSL (section 7). The new insight here is that CPS transforming a program generator allows it to generate code that is in CPS. This case-study demonstrates the fruitfulness of re-using existing host language features to enhance our embedded DSL.

In section 8, we describe our experience in using the DSL, and conclude in section 9.

In addition to the contributions above, the present work significantly extends the LMS framework, which is beneficial to future DSL efforts taking the JavaScript work as a case study. Previous LMS embeddings had to define each staged operation explicitly (like in section 2.2). This paper contributes lifting of whole traits or classes (through the `repProxy` mechanism described and used in sections 3.1& 6.3), untyped or optionally typed operations (sections 3.2, 3.3& 3.4), and typed object literals (section 6.2) including necessary language support of the Scala-Virtualized [24] compiler.

[1] Surely, the embedded language is not *exactly* JavaScript: it naturally is a subset of Scala, the host language. However, it is quite close to JavaScript. Often one can take snippets of JavaScript code and use them in the DSL with minor syntactic tweaking, as demonstrated by the Snowflake example described in section 4.

2 Introduction to LMS

In LMS, a DSL is split into two parts, its interface and its implementation. Both parts can be assembled from components in the form of Scala traits. DSL programs are written in terms of the DSL interface only, without knowledge of the implementation.

Part of each DSL interface is an abstract type constructor `Rep[_]` that is used to wrap types in the DSL programs. The DSL implementation provides a concrete instantiation of `Rep` as IR nodes. When the DSL program is staged, it produces an intermediate representation (IR), from which the final code can be generated. In the DSL program, wrapped types such as `Rep[Int]` represent staged computations while expressions of plain unwrapped types (`Int`, `Bool`, etc.) are evaluated at staging time as in [5,18].

Consider the difference between these two programs:

```
def prog1(b: Bool, x: Rep[Int]) = if (b) x else x+1
def prog2(b: Rep[Bool], x: Rep[Int]) = if (b) x else x+1
```

The only difference in these two programs is the type of the parameter `b`, illustrating that staging is purely type-driven with no syntactic overhead as the body of the programs are identical.

In `prog1`, `b` is a simple boolean, so it must be provided at staging time, and the **`if`** is evaluated at staging time. For example, `prog1(true, x)` evaluates to `x`. In `prog2`, `b` is a staged value, representing a computation which yields a boolean. So `prog2(b, x)` evaluates to an IR node for the **`if`**: `If(b, x, Plus(x, Const(1)))`.

For `prog2`, notice that the **`if`** got transformed into an IR node. To achieve this, LMS uses Scala-Virtualized [24], a suite of minimal extensions to the regular Scala compiler, in which control structures such as **`if`** can be reified into method calls, so that alternative implementations can be provided. In our case, we provide an implementation of **`if`** that constructs an IR node instead of acting as a conditional. In addition, the `+` operation is overloaded to act on both staged and unstaged expressions. This is achieved by an implicit conversion from `Rep[Int]` to a class `IntOps`, which defines a `+` method that creates an IR node `Plus` when executed. Both of `Plus`'s arguments must be staged. We use an implicit conversion to stage constants when needed by creating a `Const` IR node.

2.1 Example: A DSL Program and Its Generated JavaScript Code

The following DSL snippet creates an array representing a table of multiplications:

```
def test(n: Rep[Int]): Rep[Array[Int]] =
  for (i <- range(0, n); j <- range(0, n)) yield i*j
```

Here is the JavaScript code generated for this snippet:

```
function test(x0) {
  var x6 = []
```

```
  for(var x1=0;x1<x0;x1++){
    var x4 = []
    for(var x2=0;x2<x0;x2++){
      var x3 = x1 * x2
      x4[x2]=x3
    }
    x6.splice.apply(x6, [x6.length,0].concat(x4))
  }
  return x6
}
```

The generated code resembles single-assignment form. The nested **for**-loop is desugared into a `flatMap` which generates the nested **for**-loop and the `splice` pattern concatenating the inner `x4` arrays into one `x6` array in the JavaScript code.[2]

2.2 Walkthrough: Defining a DSL Component

To conclude the introduction to LMS, we show how to add a component for logging in a DSL, generating JavaScript code which calls `console.log`.

We start by defining the interface:

```
trait Debug extends Base {
  def log(msg: Rep[String]): Rep[Unit]
}
```

The `Base` trait is part of the core LMS framework and provides the abstract type constructor `Rep`.

Now, we define the implementation:

```
trait DebugExp extends Debug with EffectExp {
  case class Log(msg: Exp[String]) extends Def[Unit]
  def log(msg: Exp[String]): Exp[Unit] = reflectEffect(Log(msg))
}
```

The `EffectExp` trait is part of the core LMS framework. It inherits from `BaseExp` which instantiates `Rep` as `Exp`. `Exp` represents an IR via two subclasses: `Const` for constants and `Sym` for named values defining a `Def`. `Def` is the base class for all IR nodes. In our `DebugExp` trait, we extend `Def` to support a new IR node: `Log`.

IR nodes are defined as `Def`s but they are never referenced explicitly as such. Instead each `Def` has a corresponding symbol (an instance of `Sym`). IR nodes refer to each other using their symbols. This is why, in the code shown, the `msg` parameter is of type `Exp` (not `Def`). The method `log` returns an `Exp`. Calling `reflectEffect` is what creates this symbol from the `Def`.

In general, the framework provides an implicit conversion from `Def` to `Exp`, which performs common subexpression elimination by re-using the same symbol for identical definitions. We do not use the automatic conversion here, because

[2] Obviously, the generated code can be optimized further.

`log` is a side-effecting operation, and we do not want to (re)move any such calls even if their message is the same.

The framework schedules the code generation from the graph of `Exp`s and their dependencies through `Def`s. It chooses which `Sym`/`Def` pairs to emit and in which order. To implement code generation to JavaScript for our logging IR node, we simply override `emitNode` to handle `Log`:

```
trait JSGenDebug extends JSGenEffect {
  val IR: DebugExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any])(
    implicit stream: PrintWriter) = rhs match {
      case Log(s) => emitValDef(sym, "console.log(" + quote(s) + ")")
      case _ => super.emitNode(sym, rhs)
    }
}
```

Notice that in order to compose nicely with other traits, the overridden method just handles the case it knows and delegates to other traits, via **super**, the emitting of nodes it doesn't know about.

3 Gradual Typing for Interfacing with Existing APIs

Since our DSL is embedded in Scala, it inherits its static type system. However, the generated JavaScript code doesn't need the static types. Therefore, to help integrate external JavaScript libraries and APIs (for example, the browser's DOM API), we support a form of gradual typing. This has proved especially useful for rapid-prototyping, where external libraries are first incorporated dynamically, and later declared as typed APIs. Various practical and theoretical aspects of gradual typing have been studied by [38,2,1,31,32].

3.1 Typed APIs

First, we show how to incorporate an external JavaScript API in a fully-typed way into our DSL. As an example, consider the following DSL snippet, which gets the context of an HTML5 canvas element selected by id:

```
val context = document.getElementById("canvas").as[Canvas].getContext()
```

At the DSL interface level, we declare our typed APIs as abstract Scala traits:

```
trait Dom {
  val document: Rep[Element]
  trait Element
  trait ElementOps {
    def getElementById(id: Rep[String]): Rep[Element]
  }
  trait Canvas extends Element
```

```
  trait CanvasOps extends ElementOps {
    def getContext(context: Rep[String]): Rep[Context]
  }
  trait Context
  trait ContextOps {
    def lineTo(x: Rep[Int], y: Rep[Int]): Rep[Unit]
    // etc.
  }
}
```

Notice that `document` has type `Rep[Element]`, and needs to implement the interface of `ElementOps`, so that `document.getElementById("canvas")` is well-typed. We achieve this using an implicit conversion from `Rep[Element]` to `ElementOps`. At the DSL implementation level, the `ElementOps` returned by this implicit conversion needs to generate an IR node for each method call, as shown in the walkthrough in section 2.2. For example, `document.getElementById("canvas")` becomes the IR node `MethodCall(document, "getElementById", List("canvas"))`. This is a mechanical transformation, implemented by `repProxy`, a library method using reflection to intercept method calls and generate IR nodes based on the method name and the arguments of the invocation. Note that this use of reflection is purely at staging time, so there is no overhead in the generated code.

```
trait DomLift extends Dom with JSProxyBase {
  implicit def repToElementOps(x: Rep[Element]): ElementOps =
    repProxy[Element,ElementOps](x)
  implicit def repToCanvasOps(x: Rep[Canvas]): CanvasOps =
    repProxy[Canvas,CanvasOps](x)
  implicit def repToContextOps(x: Rep[Context]): ContextOps =
    repProxy[Context,ContextOps](x)
}
```

Note also that since `getElementById` returns an arbitrary DOM element, we need to cast it to a `Canvas` using `as[Canvas]`. The `as` operation is implemented simply as a cast in the host language (no IR node is created):

```
  trait AsRep {
    def as[T]: Rep[T]
  }
  implicit def asRep(x: Rep[_]): AsRep = new AsRep {
    def as[T]: Rep[T] = x.asInstanceOf[Rep[T]]
  }
```

Instead of this no-op implementation, it is possible to insert run-time check-cast assertions in the generated JavaScript code.

3.2 Casting and Optional Runtime Type Checks

The need for casting arises in a few contexts. One of them is the boundary between typed and untyped portions of a program [38,2]. Passing a value from

an untyped portion to a typed one usually requires a cast. Another situation where casts are needed is interaction with external services. For example, to process data from an external service such as Twitter, we cast it to its expected type (an array of JSON objects, each with a field called `text`):

```
type TwitterResponse = Array[JSLiteral {val text: String}]
def fetchTweets(username: Rep[String]) = {
 val raw = ajax.get { ... }
 raw.as[TwitterResponse]
}
```

A more complete example is provided in section 7. In this situation, it is useful to generate runtime checks either as an aid during development and debugging time or as a security mechanism that validates data coming from an external source. In the example above, if runtime checks are enabled, by failing early, we obtain a guarantee that all data returned from `fetchTweets` conforms to type `TwitterResponse` which means that any later access to the `text` field of any element of the data array will never fail, and always return a string.

Notice that when a typed API is defined for an external library, there are implicit casts introduced for argument and return types of the defined methods. These casts can also be checked at runtime to ensure compliance.

We have implemented a component that generates JavaScript code that asserts casts at runtime. It was fairly straightforward as the host language allows us to easily inspect types involved in casting. Since this component just provides a different implementation of the same casting method `as`, it can be enabled selectively for performance reasons.

3.3 Scala Dynamic

Since our DSL compiles to JavaScript, which is dynamically typed, it is appealing to allow expressions and APIs in our DSL to also, selectively, be dynamically typed. This is especially useful for rapid-prototyping.

We provide a component, `JSDynamic`, which allows any expression to become dynamically typed, by wrapping it in a `dynamic` call. The `dynamic` wrapper returns a `DynamicRep`, on which any method call, field access and field update is possible. A dynamic method call and field access returns another `DynamicRep` expression, so dynamic expressions can be chained.

`DynamicRep` exploits a new Scala feature[3], based on a special trait `Dynamic`: An expression whose type `T` is a subtype of `Dynamic` is subject to the following rewrites:

- `x.f` rewrites to `x.selectDynamic("f")`,
- `x.m(a, ..., z)` rewrites to `x.applyDynamic("m")(a, ..., z)`,
- `x.f = v` rewrites to `x.updateDynamic("f")(v)`.

These rewrites take place when the type `T` doesn't statically have field `f` and method `m`.

[3] C#'s type "dynamic" [3] can serve the same purpose.

At the implementation level, as these rewriting take place, we generate IR nodes which allow us to then generate straightforward JavaScript for the original expressions. For example, for the expression `dynamic(x).foo(1, 2)`, we would generate an IR node like `MethodCall(x, "foo", List(Const(1), Const(2)))`. From this IR node, it is easy to generate the JavaScript code `x.foo(1, 2)`. Note the similarity with the IR nodes generated for typed APIs.

3.4 From Dynamic to Static

This possibility to escape into dynamic typing is particularly useful in simplifying the incorporation of external JavaScript APIs and libraries. Sometimes, the user might not want to build a statically typed API for each external JavaScript library. In addition, for some library, it might be awkward to come up with such a statically-typed interface. In general, we expect users to start with a dynamic API for an external library, and progressively migrate it to a typed API as the code matures. Consider again the example introduced in the typed API section:

```
val context = document.getElementById("canvas").as[Canvas].getContext()
```

In a fully dynamic scenario, we declare the DOM API simply as:

```
trait Dom extends JSDynamic {
  val document: DynamicRep
}
```

The `as[Canvas]` cast is not necessary in this dynamically typed setting:

```
val context = document.getElementById("canvas").getContext()
```

As a first step towards statically typing the API, we declare the type `Element`:

```
trait Dom extends JSProxyBase with JSDynamic {
  val document: Rep[Element]
  trait Element
  trait ElementOps {
    def getElementById(id: Rep[String]): DynamicRep
  }
  implicit def repToElementOps(x: Rep[Element]): ElementOps =
    repProxy[Element,ElementOps](x)
}
```

Since the method `getElementById` returns a `DynamicRep`, only the emphasized part of the expression is statically typed:

```
val context = document.getElementById("canvas").getContext()
```

We can then complete the static typing by declaring types for `Canvas` and `Context` as seen in the typed API section.

In our gradual typing scheme, an expression is either completely statically typed or completely dynamically typed. Once `document` is declared as `Rep[Element]`

instead of `DynamicRep`, it is a type error to call an arbitrary method which is not part of its declared `ElementOps` interface. If needed, it is always possible to explicitly move from a statically-typed expression to a dynamically-typed one by wrapping it in a `dynamic` call.

4 Sharing Code between Client and Server

In addition to generating JavaScript / client-side code, we want to be able to re-use our DSL code on the Scala / server-side. In the LMS approach, the DSL uses the abstract type constructor `Rep` [23]. When generating JavaScript, this abstract type constructor is defined by IR nodes. Another definition, which we dub "trivial embedding", is to use the identity type constructor: `Rep[T] = T`. By stripping out `Rep`s in this way, our DSL can operate on concrete Scala types, replacing staging with direct evaluation. Even in the trivial embedding, when the DSL code operates on concrete Scala types, virtualization still occurs because the usage layer of the DSL is still in terms of abstract `Rep`s.

As an example, consider the following DSL snippet, which computes the absolute value:

```
trait Ex extends JS {
  def abs(x: Rep[Int]) = if (x < 0) -x else x
}
```

We can use this DSL snippet to generate JavaScript code:

```
new Ex with JSExp { self =>
  val codegen = new JSGen { ... }
  codegen.emitSource(abs _, "abs", ...)
}
```

We can also use this DSL snippet directly in Scala via the trivial embedding (defined by `JSInScala`):

```
new Ex with JSInScala { self =>
  println(abs(-3))
}
```

In the JavaScript example (when mixing in `JSExp`) evaluating `abs(x)` results in an IR tree roughly equivalent to `If(LessThan(Sym("x"), Const(0)), Neg(Sym("x")), Sym("x"))`. In the trivial embedding, when `abs(-3)` gets called, it evaluates to `3` by executing the virtualized **`if`** as a normal condition. In short, in the trivial embedding, the virtualized method calls are evaluated in-place without constructing IR nodes.

In the previous section, we showed how to define typed APIs to represent JavaScript external libraries or dependencies. In the trivial embedding, we need to give an interpretation to these APIs. For example, we can implement a Canvas context in Scala by using native graphics to draw on a desktop widget instead of a web page. We translated David Flanagan's Canvas example from the book

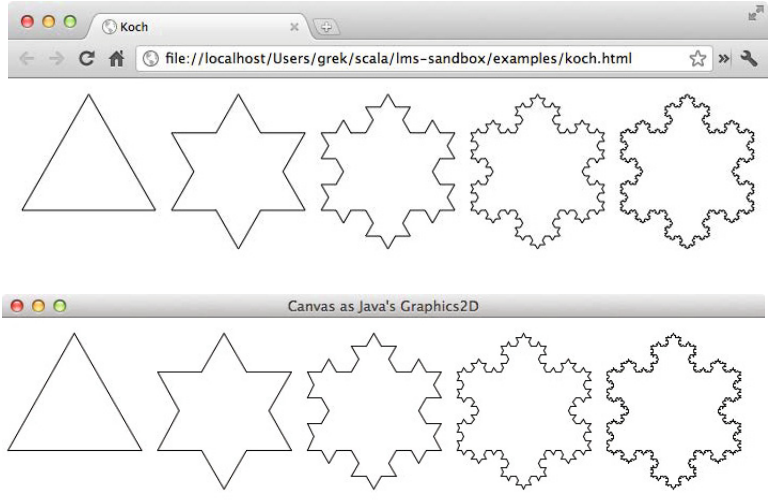


Fig. 1. Snowflakes rendered using HTML5 Canvas and Java's 2D

"JavaScript: The Definitive Guide" [16], which draws Koch snowflakes on a canvas [15]. First, the translation from JavaScript to our DSL is straightforward: the code looks the same except for some minor declarations. Then, from our DSL code, we can generate JavaScript code to draw the snowflakes on a canvas as in the original code. In addition, via the trivial embedding, we can execute the DSL code in Scala to draw snowflakes on a desktop widget. Screenshot presenting snowflakes rendered in a browser using HTML5 Canvas and Java's 2D are presented in figure 1.

HTML5 Canvas is a standard that is not implemented by all browsers yet so a fall-back mechanism is needed to support users of older browsers. This can be achieved through the trivial embedding by drawing using Java's 2D API, saving the result as an image and sending it to the browser. The decision to either generate a JavaScript snippet that draws on canvas and send it to the browser, or render the image on the server can be made at runtime (e.g. after inspecting information about the client's browser). In the case of rendering on the server-side, one can store computation that renders an image using Java's graphics 2D in a hash map and send back to the client the key as an url for an image. When a browser makes a second request, computation can be restored from the hash map, executed and the result sent back to the browser. All of that is possible because the computation itself is expressed against an abstract DSL API so we can swap implementations to either generate JavaScript or run the computation at the server side. Moreover, our DSL is just a library and computations are expressed as first-class values in a host language so they can be passed around the server program, stored in a hash map, etc.

A drawback of the trivial embedding is that although lightweight, virtualization is not completely free, and a small virtualization overhead is incurred each time the program is evaluated. To avoid this, we could generate Scala code in

the same way as we now generate JavaScript code, relying on a mechanism to incorporate the generated Scala code into the rest of the program. In fact, this approach is taken by the Delite framework.

5 Modularity Interlude

The design of our DSL supports modularity at many levels. We use Scala's traits heavily to allow our DSL to be assembled from and extended with components [25].

For example, the feature to escape into dynamic typing is implemented as an independent component that can be mixed and matched with others. Similarly, users specify external APIs as components. The separation between the interface level and the implementation level is also done by having distinct components for each level. This allows the same DSL program to be interpreted in multiple ways, as has been shown with the trivial embedding to Scala.

As the features available in a DSL program are specified by composing components, it is possible to use this mechanism to enforce that a subprogram only uses a restricted set of features. For example, worker threads in JavaScript ("WebWorkers") are not allowed to manipulate the DOM. This can be enforced by not mixing in the DOM component in the subprogram for a worker thread.

The code generation level is assembled from components as in the interface and implementation levels. Furthermore, optimizations in code generation can be implemented as optional components to be mixed in.

6 Reification of Objects

By exploiting staging, the generated code can remain simple and relatively unstructured as many of the high-level constructs can be evaluated away at staging time. However, it is sometimes useful to be able to reify more complex structures. For example, APIs sometimes expect arguments or return results as object literals. Therefore, we support a few type-safe ways to create more complex staged structures, which we explain below.

6.1 Functions

A function in the host language acting on reified types has type: `Rep[A] => Rep[B]`. For example, the function `inc` has type `Rep[Int] => Rep[Int]`:

```
val inc = (x: Rep[Int]) => x + 1
```

Invoking such a function at staging time simply inlines the call: `inc(4*x)` results in `4*x + 1`. This is useful and nice, because it removes abstraction overhead from the generated code.

We also want the ability to treat functions as first-class staged values, since JavaScript supports them. In order to do this, we provide a higher-order function

`fun` which takes a function of type `Rep[A] => Rep[B]` and converts[4] it to a staged function of type `Rep[A => B]`. For example, if we define `inc` in the following way, its type is `Rep[Int => Int]`:

```
val inc = fun { (x: Rep[Int]) => x + 1 }
```

Calling `inc(4*x)` results in an `Apply` IR node. We actually generate JavaScript code for the staged `inc` function, while we did not for the unstaged one, since it is inlined at every call site during staging. The generated code looks roughly like the following:

```
var inc = function(a) {
  return a+1
}
inc(4*x)
```

First-class functions are widely used in JavaScript. One particular common case is for callback-driven programming. Therefore, staged functions are important to interface with existing libraries. They will also play a crucial role in section 7, where we abstract over callback-driven programming.

6.2 Typed Object Literals

Our DSL provides typed immutable object literals to represent JavaScript object literals. As an example:

```
val o = new JSLiteral {
  val a = 1
  val b = a + 1
}
```

`o` has type `Rep[JSLiteral {val a: Int; val b: Int}]`. All the fields of `o` are `Rep`s, so `o.a` has type `Rep[Int]`. The translation to JavaScript is straightforward:

```
var o = { a : 1, b : 2 }
```

This straightforward translation makes it possible to pass the typed object literals of our DSL to JavaScript functions which expects object literals, such as `JSON.stringify` or jQuery's `css`.

As for implementation, notice that the type of a `new JSLiteral {...}` expression is not `JSLiteral {...}` but `Rep[JSLiteral {...}]`. This is achieved with support from the Scala-Virtualized compiler. `JSLiteral` inherits from a special trait, which indicates its `new` expressions should be reified. So the `new` expression is turned into a method call, with information about all the field definitions. A complication is that a field definition might reference another field being defined (such as `b` being defined in terms of `a` in the example above). So each definition is represented by its name and a *function* which takes a self type for the object

[4] We refer the reader to [28] for implementation details.

literal. These definition functions are evaluated in an order which allows the self references to be resolved.

6.3 Classes

Our DSL also supports reified classes. For convenience, these are defined as traits, use the `repProxy` mechanism underlying typed APIs and are also implemented using reflection. The translation to JavaScript, based on constructors and prototypes, is straightforward. Through the trivial embedding, classes implemented in our DSL can be used on both the server and client sides.

7 CPS Transformation for Asynchronous Code Patterns

A callback-driven programming style is pervasive in JavaScript programs. Because of lack of thread support, callbacks are used for I/O, scheduling and event-handling. For example, in an Ajax call, one has to provide a callback that will be called once the requested data arrives from the server. This style of programming is known to be unwieldy in more complicated scenarios. To give a specific example, let's consider a scenario where we have an array of Twitter account names and we want to ask Twitter for the latest tweets of each account. Once we obtain the tweets of all users, we would like to log that fact in a console.

We'll implement this program both in JavaScript and in our DSL. Let's start by implementing logic that fetches tweets for a single user by using the jQuery library for Ajax calls (listings 1.1 & 1.2).

Listing 1.1. Fetching tweets in JavaScript

```
function fetchTweets(username, callback) {
  jQuery.ajax({
    url:"http://api.twitter.com/1/
       statuses/user_timeline.json/",
    type: "GET",
    dataType: "jsonp",
    data: {
      screen_name : username,
      include_rts : true,
      count : 5,
      include_entities : true
    },
    success: callback
  })
}
```

Listing 1.2. Fetching tweets in DSL

```
def fetchTweets(username: Rep[String]) =
  (ajax.get {
    new JSLiteral {
    val url = "http://api.twitter.com/1/
       statuses/user_timeline.json"
    val `type` = "GET"
    val dataType = "jsonp"
    val data = new JSLiteral {
      val screen_name = username
      val include_rts = true
      val count = 5
      val include_entities = true
    }
  }
}).as[TwitterResponse]

type TwitterResponse =
  Array[JSLiteral {val text: String}]
```

Note that JavaScript version takes a callback as second argument that will be used to process the fetched tweets. We provide the rest of the logic that iterates over an array of users and makes Ajax requests (listings 1.3 & 1.4).

Listing 1.3. Twitter example in JavaScript

```
var processed = 0
var users = ["gkossakowski", "odersky",
  "adriaanm"]
users.forEach(function (user) {
  console.log("fetching " + user)
  fetchTweets(user, function(data) {
    console.log("finished fetching " + user)
    data.forEach(function (tweet) {
      console.log("fetched " + tweet.text)
    })
    processed += 1
    if (processed == users.length) {
      console.log("done")
    }
  })
})
```

Listing 1.4. Twitter example in DSL

```
val users = array("gkossakowski", "odersky",
  "adriaanm")
for (user <- users.parSuspendable) {
  console.log("fetching " + user)
  val tweets = fetchTweets(user)
  console.log("finished fetching " + user)
  for (t <- tweets)
    console.log("fetched " + t.text)
}
console.log("done")
```

Because of the inverted control flow of callbacks, synchronization between callbacks has to be handled manually. Also, the inverted control flow leads to a code structure that is distant from the programmer's intent. Notice that the in JavaScript version, the call to console that prints "done" is put inside of the foreach loop. If it was put it after the loop, we would get "done" printed *before* any Ajax call has been made leading to counterintuitive behaviour.

As an alternative to the callback-driven programming style, one can use an explicit monadic style, possibly sugared by a Haskell-like "do"-notation. This is the approach taken by F# Web Tools [26]. However, this requires rewriting possibly large parts of a program into monadic style when a single async operation is added. Another possibility is to automatically transform the program into continuation passing style (CPS), enabling the programmer to express the algorithm in a straightforward, sequential way and creating all the necessary callbacks and book-keeping code automatically. Links [10] uses this approach. However, a whole-program CPS transformation can cause performance degradation, code size blow-up, and stack overflows. In addition, it is not clear how to interact with existing non-CPS libraries as the whole program needs to adhere to the CPS style. We suggest using a *selective* CPS transformation, which precisely identifies what needs to be CPS transformed.

In fact, the Scala compiler already does selective CPS transformations of Scala programs, driven by `@suspendable` type annotations [27,12,13]. We show how this mechanism can be used for transforming our DSL code before staging and stripping out most CPS abstractions at staging time. The generated JavaScript code does not contain any CPS-specific code but is written in CPS-style by use of JavaScript anonymous functions.

Our implementation to support continuations is an example of an interesting technique of applying selective cps transformation to embedded DSLs by means of a deep linguistic reuse, exploiting a host language feature to implement the corresponding DSL feature without (much) additional work. An interesting

insight is that our method of CPS transforming a direct-style program generator allows it to generate code that is in CPS.

7.1 CPS in Scala

Before presenting how CPS transformations are used in our DSL, let's consider a typical situation where CPS rewrites act on Scala programs.

As an example, we consider a `sleep` method implemented in non-blocking, asynchronous style. This is useful, for example, when using `ThreadPool`s as no thread is being blocked during the sleep period. Let's see how our `sleep` method written in CPS can be used:

```
def foo() = {
  sleep(1000)
  println("Called foo")
}
reset {
  println("look, Ma ...")
  foo()
  sleep(1000)
  println(" no threads!")
}
```

The `reset` delimits the scope of CPS rewrite. Let's see how the rewrite itself looks like:

```
def foo(): ControlContext = {
  sleep(1000).map((tmp1: Unit) => println("Called foo"))
}
reset {
  println("look, Ma ...")
  foo().flatMap((tmp2: Unit) =>
    sleep(1000).map((tmp3: Unit) => println(" no threads!"))
  )
}
```

There are a few things worth noting here. First, the return type of `foo` method is rewritten to be `ControlContext`.[5] This is due to the fact that `sleep` is used in the body of `foo`. Also, note that the code to be executed after sleeping is captured as a continuation (anynomous function) and passed to the `ControlContext` through a call of the `map` method. The body of the `reset` block is rewritten in a similar vein.

Now, let's have a closer look how `sleep` itself is implemented:

[5] The `ControlContext` class implements the continuation monad and is provided by Scala's standard library. `ControlContext[A,B,C]` is similar to C#'s `Task<T>`, but more general. In our discussion, we omit the type parameters for simplicity.

Listing 1.5. `sleep` implementation

```
import java.util.{Timer,TimerTask}
val timer = new Timer
def sleep(delay: Int): Unit @suspendable = shift { k =>
  val task = new TimerTask { def run() = k() }
  timer.schedule(task, delay.toLong)
}
```

Notice the `@suspendable` type annotation attached to `sleep`'s return type `Unit`. The `@suspendable` annotation means that the `sleep` method can be used in a side-effecting continuation context. In the definition of the `sleep` method, we use Java's `Timer` and `TimerTask` abstractions for asynchronous, delayed task execution. The `TimerTask` interface has one method, `run`, that will be executed after a specified period of time. The `shift` control abstraction allows us to capture the continuation as a first-class value and then use it in the body of the `run` method. Both the `reset` and `shift` control abstractions are described in detail in [12].

After the CPS transformation, the code presented above becomes

```
def sleep(time: Int): ControlContext =
  shiftR { k =>
    val task = new TimerTask { def run() = k() }
    timer.schedule(task, delay.toLong)
  }
```

After the rewriting, all CPS-related type annotations are dropped and use of the `ControlContext` class that supports continuation passing is introduced. The `shiftR` method is an internal method that takes a block (which itself is a function) and wraps it in `ControlContext` structure.

7.2 CPS and Staging

Let's write the example from listing 1.5 in our DSL. We need to define `sleep` to use JavaScript's `setTimeout` [6] as a replacement for the `Timer` abstraction.

```
def sleep(delay: Rep[Int]) = shift { k: (Rep[Unit]=>Rep[Unit]) =>
  window.setTimeout(fun(k), delay)
}
```

The `setTimeout` method expects an argument of type `Rep[Unit=>Unit]` which denotes a *representation* of a function of type `Unit=>Unit`. The `shift` method offers us a function of type `Rep[Unit] => Rep[Unit]`, so we need to reify it to obtain the desired representation. The reification is achieved by the `fun` function (described in 6.1) provided by our framework and performed at staging time.

It is important to note that reification preserves function composition (roughly speaking it is a homomorphism between DSL code and generated JavaScript code).

[6] The `setTimeout` function asynchronously executes a function passed as argument after a specified delay.

Specifically, let `f: Rep[A] => Rep[B]` and `g: Rep[B] => Rep[C]` then `fun(g compose f) == (fun(g) compose fun(f))` where we consider two reified functions to be equal if they yield the same observable effects at runtime [7]. That property of function reification is at the core of reusing the continuation monad in our DSL. Thanks to the fact that the continuation monad composes functions, we can just insert reification at some places (like in a `sleep`) and make sure that we reify *effects* of the continuation monad without the need to reify the monad itself.

7.3 CPS for Suspendable Traversals

We need to be able to suspend our execution while traversing an array in order to implement functionality from listing 1.4. Let's consider a simplified example where we want to iterate over an array and sleep during each iteration. We present both code written in JavaScript and our DSL that achieves that (listings 1.6 & 1.7).

Listing 1.6. JavaScript

```
var xs = [1, 2, 3]
var i = 0
var msg = null
function f1() {
  if (i < xs.length) {
    window.setTimeout(f2, xs[i]*1000)
    msg = xs[i]
    i++
  } else {
    console.log("done")
  }
}
function f2() {
  console.log(msg)
  f1()
}
f1()
```

Listing 1.7. DSL

```
val xs = array(1, 2, 3)
// shorthand for xs.suspendable.foreach
for (x <- xs.suspendable) {
  sleep(x * 1000)
  console.log(String.valueOf(x))
}
log("done")
```

Both programs, when executed, will print the following to the JavaScript console:

```
//pause for 1s
1
//pause for 2s
2
//pause for 3s
3
done
```

In the DSL code, we use a `suspendable` variant of arrays, which is achieved through an implicit conversion from regular arrays:

[7] Note that composition on the left and the right hand side of the equation is not the same operation but they have the same observable runtime effect.

```
implicit def richArray(xs: Rep[Array[A]]) = new {
  def suspendable: SArrayOps[A] = new SArrayOps[A](xs)
}
```

The idea behind `suspendable` arrays is that iteration over them can be suspended. We'll have a closer look at how to achieve that with the help of CPS. The `suspendable` method returns a new instance of the `SArrayOps` class defined here:

Listing 1.8. Suspendable foreach

```
class SArrayOps(xs: Rep[Array[A]]) {
  def foreach(yld: Rep[A] => Rep[Unit] @suspendable):
    Rep[Unit] @suspendable = {
      var i = 0
      suspendableWhile(i < xs.length) { yld(xs(i)); i += 1 }
    }
}
```

Note that one cannot use while loops in CPS but we can simulate them with recursive functions. Let's see how a regular while loop can be simulated with a recursive function with call-by-name parameters:

```
def recursiveWhile(cond: => Boolean)(body: => Unit): Unit = {
  def rec = () => if (cond) { body; rec() } else {}
  rec()
}
```

By adding CPS-related declarations and control abstractions, we implement `suspendableWhile`:

```
def suspendableWhile(cond: => Rep[Boolean])(
  body: => Rep[Unit] @suspendable): Rep[Unit] @suspendable =
  shift { k =>
    def rec = fun { () =>
      if (cond) reset { body; rec() } else { k() }
    }
    rec()
  }
```

7.4 Defining the Ajax API

With the abstractions for suspendable loops and traversals at hand, what remains to complete the Twitter example from the beginning of the section is the actual Ajax request/response cycle.

The Ajax interface component provides a type `Request` that captures the request structure expected by the underlying JavaScript/jQuery implementation and the necessary object and method definitions to enable the use of `ajax.get` in user code:

```
trait Ajax extends JS with CPS {
  type Request = JSLiteral {
    val url: String
    val `type`: String
    val dataType: String
    val data: JSLiteral
  }
  type Response = Any
  object ajax {
    def get(request: Rep[Request]) = ajax_get(request)
  }
  def ajax_get(request: Rep[Request]): Rep[Response] @suspendable
}
```

Notice that the `Request` type is flexible enough to support an arbitrary object literal type for the `data` field through subtyping. The `Response` type alias points at `Any` which means that it is the user's responsibility to either use `dynamic` or perform an explicit cast to the expected data type.

The corresponding implementation component implements `ajax_get` to capture a continuation, reify it as a staged function using `fun` and store it in an `AjaxGet` IR node.

```
trait AjaxExp extends JSExp with Ajax {
  case class AjaxGet(request: Rep[Request],
    success: Rep[Response => Unit]) extends Def[Unit]
  def ajax_get(request: Rep[Request]): Rep[Response] @suspendable =
    shift { k =>
      reflectEffect(AjaxGet(request, fun(k)))
    }
}
```

During code generation, we emit code to attach the captured continuation as a callback function in the `success` field of the request object:

```
trait GenAjax extends JSGenBase {
  val IR: AjaxExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any])(
    implicit stream: PrintWriter) = rhs match {
      case AjaxGet(req, succ) =>
        stream.println(quote(req) + ".success = " + quote(succ))
        emitValDef(sym, "jQuery.ajax(" + quote(req) + ")")
    case _ => super.emitNode(sym, rhs)
  }
}
```

It is interesting to note that, since the request already has the right structure for the `jQuery.ajax` function, we can simply pass it to the framework-provided `quote` method, which knows how to generate JavaScript representations of any `JSLiteral`.

The Ajax component completes the functionality required to run the Twitter example with one caveat: The outer loop in listing 1.4 uses `parSuspendable` to traverse arrays instead of the `suspendable` traversal variant we have defined in listing 1.8.

In fact, if we change the code to use `suspendable` instead of `parSuspendable` and run the generated JavaScript program, we will get following output printed to the JavaScript console:

```
fetching gkossakowski
finished fetching gkossakowski
fetched [...]
fetched [...]
fetching odersky
finished fetching odersky
fetched [...]
fetched [...]
fetching adriaanm
finished fetching adriaanm
fetched [...]
fetched [...]
done
```

Notice that all Ajax requests were done sequentially. Specifically, there was just one Ajax request active at a given time; when the callback to process one request is called, it would resume the continuation to start another request, and so on. In many cases this is exactly the desired behavior, however, we will most likely want to perform our Ajax request in parallel.

7.5 CPS for Parallelism

The goal of this section is to implement a parallel variant of the `foreach` method from listing 1.8. We'll start with defining a few primitives like futures and dataflow cells. Let's start with cells, which we decide to define in JavaScript, as another example of integrating external libraries with our DSL:

Listing 1.9. JavaScript-based implementation of a non-blocking Cell

```
function Cell() {
  this.value = undefined
  this.isDefined = false
  this.queue = []
  this.get = function (k) {
    if (this.isDefined) {
```

```
      k(this.value)
    } else {
      this.queue.push(k)
    }
  }
  this.set = function (v) {
    if (this.isDefined) {
      throw "can't set value twice"
    } else {
      this.value = v
      this.isDefined = true
      this.queue.forEach(function (f) {
        f(v) //non-trivial spawn could be used here
      })
    }
  }
}
```

A cell object allows us to track dependencies between values. Whenever the `get` method is called and the value is not in the cell yet, the continuation will be added to a queue so it can be suspended until the value arrives. The `set` method takes care of resuming queued continuations. We expose `Cell` as an external library using our typed API mechanism and we use it for implementing an abstraction for futures.

```
def createCell(): Rep[Cell[A]]
trait Cell[A]
trait CellOps[A] {
  def get(k: Rep[A => Unit]): Rep[Unit]
  def set(v: Rep[A]): Rep[Unit]
}
implicit def repToCellOps(x: Rep[Cell[A]]): CellOps[A] =
  repProxy[Cell[A],CellOps[A]](x)

def spawn(body: => Rep[Unit] @suspendable): Rep[Unit] = {
  reset(body) //non-trivial implementation uses
              //trampolining to prevent stack overflows
}
def future(body: => Rep[A] @suspendable) = {
  val cell = createCell[A]()
  spawn { cell.set(body) }
  cell
}
```

The last bit of general functionality we need is `RichCellOps` that ties `Cells` and continuations together inside of our DSL.

```
class RichCellOps(cell: Rep[Cell[A]]) {
  def apply() = shift { k: (Rep[A] => Rep[Unit]) =>
    cell.get(fun(k))
  }
}
implicit def richCellOps(x: Rep[Cell[A]]): RichCell[A] =
  new RichCellOps(x)
```

It is worth noting that `RichCellOps` is not reified so it will be dropped at staging time and its method will get inlined whenever used. Also, it contains CPS-specific code that allows us to capture the continuation. The `fun` function reifies the captured continuation.

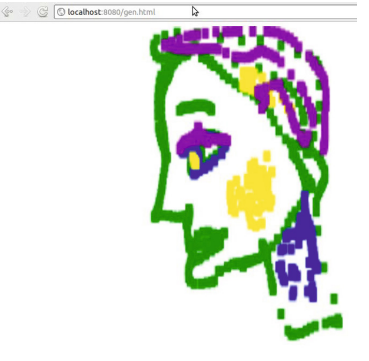
We are ready to present the parallel version of `foreach` defined in listing 1.8.

```
def foreach(yld: Rep[A] => Rep[Unit] @suspendable):
  Rep[Unit] @suspendable = {
    val futures = xs.map(x => future(yld(x)))
    futures.suspendable.foreach(_.apply())
  }
```

We instantiate each future separately so they can be executed in parallel. As a second step we make sure that all futures are evaluated before we leave the `foreach` method by forcing evaluation of each future and "waiting" for its completion. Thanks to CPS transformations, all of that will be implemented in a non-blocking style.

The only difference between the parallel and serial versions of the Twitter example 1.4 is the use of `parSuspendable` instead of `suspendable` so the parallel implementation of the `foreach` method is used. The rest of the code stays the same. It is easy to switch between both versions, and users are free to make their choice according to their needs and performance requirements.

8 Evaluation



We have implemented our DSL in Scala.[8] We used our DSL to develop a few web applications, which are simple but not trivial. First, we implemented a few drawing examples like the snowflakes of figure 1. We extended the Twitter example from section 7, which presents the latest tweets from selected users in an interactive way. In order to do so, we incorporated a useful subset of the DOM API and jQuery library using our typed APIs.

[8] The code can be found at `http://github.com/js-scala`

Moreover, we integrated our DSL with an existing web framework, Play 2.0.[9] Our DSL can be used to define Play form validators that are executed on both the client and server sides. We managed to achieve that without deep changes to the framework, proving that our DSL can be used as a library.

Finally, we developed a collaborative drawing application, which includes a server-side component (implemented using the Jetty web server). We use web sockets to communicate between the server and clients. Each client transmits drawing commands to the server, which broadcasts them to all the clients. When a new client joins, the server sends the complete drawing history to the new client, and the client reconstructs the image by playing back the commands. A very simple improvement is to make the server execute the drawing commands as well, and keep an up-to-date bitmap of the drawing – this can easily be achieved by using the trivial embedding described in section 4. New clients then just obtain the bitmap instead of replaying the history, which can be large and grow unboundedly.

9 Conclusion

In this paper, we have shown how to embed a JavaScript DSL in Scala using LMS. A recurring theme of our approach is to exploit the features of the host language to enhance the DSL with minimal effort. Moreover, through staging, we can use many abstractions at the code-generation stage, without complexity and performance overhead in the generated code. We believe this approach addresses some important challenges of developing rich web applications.

Acknowledgments. We thank Adriaan Moors for maintaining the Scala-Virtualized compiler, adding our feature requests, fixing our reported bugs, and for insightful discussions.

References

1. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Trans. Program. Lang. Syst. 13, 237–268 (1991)
2. Ahmed, A., Findler, R.B., Matthews, J., Wadler, P.: Blame for all. In: Proceedings for the 1st Workshop on Script to Program Evolution, STOP 2009, pp. 1–13. ACM, New York (2009)
3. Bierman, G., Meijer, E., Torgersen, M.: Adding Dynamic Types to C♯. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 76–100. Springer, Heidelberg (2010)
4. Brown, K., Sujeeth, A., Lee, H., Rompf, T., Chafi, H., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. In: 20th International Conference on Parallel Architectures and Compilation Techniques, PACT (2011)
5. Carette, J., Kiselyov, O., Shan, C.-C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19, 509–543 (2009)

[9] http://www.playframework.org

6. Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M., Olukotun, K.: Language Virtualization for Heterogeneous Parallel Computing. Onward! (2010)
7. Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP (2011)
8. https://github.com/clojure/clojurescript/wiki
9. http://jashkenas.github.com/coffee-script/
10. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web Programming Without Tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)
11. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: The Essence of Form Abstraction. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 205–220. Springer, Heidelberg (2008)
12. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, pp. 151–160. ACM, New York (1990)
13. Danvy, O., Filinski, A.: Representing control: A study of the cps transformation. Mathematical Structures in Computer Science 2(4), 361–391 (1992)
14. http://www.dartlang.org/
15. Flanagan, D.: (2011), https://github.com/davidflanagan/javascript6_examples/blob/master/examples/21.06.koch.js
16. Flanagan, D.: JavaScript: The Definitive Guide, 6th edn. O'Reilly Media, Inc. (2011)
17. http://code.google.com/webtoolkit/
18. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of dsls. In: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE 2008, pp. 137–148. ACM, New York (2008)
19. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In: Rinard, M. (ed.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, Reno, NV, USA, October 17-21, pp. 444–463 (2010)
20. Khoo, Y.P., Hicks, M., Foster, J.S., Sazawal, V.: Directing javascript with arrows. SIGPLAN Not. 44, 49–58 (2009)
21. Lee, H., Brown, K., Sujeeth, A., Chafi, H., Rompf, T., Odersky, M., Olukotun, K.: Implementing domain-specific languages for heterogeneous parallel computing. IEEE Micro. 31, 42–53 (2011)
22. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for ajax applications. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 1–20. ACM, New York (2009)
23. Moors, A., Piessens, F., Odersky, M.: Generics of a Higher Kind. ACM SIGPLAN Notices 43, 423–438 (2008)
24. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. In: PEPM 2012 (2012)
25. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proceedings of OOPSLA 2005 (2005)
26. Petříček, T., Syme, D.: F# web tools: Rich client/server web applications in f#

27. Rompf, T., Maier, I., Odersky, M.: Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. ACM, New York (2009)
28. Rompf, T., Odersky, M.: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In: GPCE (2010)
29. Rompf, T., Sujeeth, A.K., Lee, H., Brown, K.J., Chafi, H., Oderksy, M., Olukotun, K.: Building-blocks for performance oriented DSLs. In: Electronic Proceedings in Theoretical Computer Science (2011)
30. http://scalagwt.github.com/
31. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
32. Siek, J.G., Taha, W.: Gradual Typing for Objects. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007)
33. Sujeeth, A.K., Lee, H., Brown, K.J., Rompf, T., Wu, M., Atreya, A.R., Odersky, M., Olukotun, K.: OptiML: an implicitly parallel domain-specific language for machine learning. In: Proceedings of the 28th International Conference on Machine Learning, ICML (2011)
34. Taha, W., Sheard, T.: Metaml and multi-stage programming with explicit annotations. Theor. Comput. Sci. 248, 211–242 (2000)
35. Ureche, V., Rompf, T., Sujeeth, A., Chafi, H., Odersky, M.: Stagedsac: A case study in performance-oriented dsl development. In: PEPM (2012)
36. Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)
37. Vogt, J.C.: Type Safe Integration of Query Languages into Scala. Diplomarbeit, RWTH Aachen, Germany (2011)
38. Wadler, P., Findler, R.B.: Well-Typed Programs Can't Be Blamed. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 1–16. Springer, Heidelberg (2009)

Correlation Tracking for Points-To Analysis of JavaScript

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip

IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{msridhar,dolby,satishchandra,mschaefer,ftip}@us.ibm.com

Abstract. JavaScript poses significant challenges for points-to analysis, particularly due to its flexible object model in which object properties can be created and deleted at run-time and accessed via first-class names. These features cause an increase in the worst-case running time of field-sensitive Andersen-style analysis, which becomes $O(N^4)$, where N is the program size, in contrast to the $O(N^3)$ bound for languages like Java. In practice, we found that a standard implementation of the analysis was unable to analyze popular JavaScript frameworks.

We identify *correlated dynamic property accesses* as a common code pattern that is analyzed very imprecisely by the standard analysis, and show how a novel *correlation tracking* technique enables us to handle this pattern more precisely, thereby making the analysis more scalable. In an experimental evaluation, we found that correlation tracking often dramatically improved analysis scalability and precision on popular JavaScript frameworks, though in some cases scalability challenges remain.

Keywords: Points-to analysis, call graph construction, JavaScript.

1 Introduction

JavaScript is rapidly gaining in popularity because it enables programmers to write rich web applications with full-featured user interfaces and portability across desktop and mobile platforms. Recently, pointer analysis for JavaScript has been used to enable applications such as security analysis [10, 12], bug finding [14], and automated refactoring [8].

Real-world web applications increasingly make use of framework libraries like *jquery*[1] that abstract away browser incompatibilities and provide advanced DOM manipulation and user interface libraries. A recent survey [27] found that more than half of all surveyed sites use one of the nine most popular frameworks. These frameworks present a formidable challenge to static analysis, since they are relatively large and make frequent use of highly dynamic language features.

In particular, JavaScript's flexible object model presents a major challenge for scalable and precise points-to analysis. In JavaScript, an *object* has zero or more *properties* (corresponding to *instance fields* in languages like Java), each

[1] http://jquery.com

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 435–458, 2012.

identified by a unique name. Properties may contain any kind of value, including first-class functions, and programmers may define a "method" on an object by assigning a function to one of its properties, as in the following example:

```
o.foo = function f1() { return 23; };
o.bar = function f2() { return 42; };
o.foo();
```

To identify the precise call target for the call `o.foo()`, an analysis must be able to compute points-to targets for `o.foo` and `o.bar` separately and not conflate them. This is usually achieved by a *field-sensitive* analysis [16].

Field-sensitive analysis for JavaScript is complicated by the fact that properties can be accessed by computed names. Consider the following code:

```
f = p() ? "foo" : "baz";
o[f] = "Hello, world!";
```

Here, `f` is assigned either the value `"foo"` or the value `"baz"`, depending on the return value of `p`. A *dynamic property access* is then used to store a string value into a property of object `o` whose name is determined by the value of `f`. If `f` is `"foo"`, the existing property `o.foo` is overwritten, otherwise a new property `o.baz` is created and initialized to the given value.

In the presence of dynamic property accesses, performing a field-sensitive analysis poses both theoretical and practical challenges. We show that, surprisingly, extending a standard implementation of field-sensitive Andersen's points-to analysis [3] to handle dynamic property accesses causes the implementation to run in worst-case $O(N^4)$ time, where N denotes the size of the program, compared to the typical $O(N^3)$ bound for other programming languages.

This increased complexity is not merely of theoretical interest—we were unable to scale a traditional field-sensitive analysis to handle JavaScript frameworks like *jquery*. These frameworks generally make heavy use of dynamic property accesses to reflectively access object properties. In combination with other features of JavaScript such as first-class functions, these operations cause an explosion in analysis imprecision that makes call graph construction intractable in practice, as we illustrate on an example in Section 2. And while these operations are most idiomatic and common in JavaScript, exactly the same operations can be written in other scripting languages like Python.

We have devised a technique that helps address issues caused by dynamic property accesses by making the points-to analysis *more precise*. We observed that for property writes that cause imprecision in practice, there is often an obvious correlation between the updated location and the stored value that is ignored by the points-to analysis. For example, for the statement `x[p] = y[p]` (which copies the value for property `p` in `y` to property `p` in `x`), a standard points-to analysis does not track the fact the same property `p` is accessed on both sides of the assignment. If `p` ranges over many possible values, this leads to conflation of many unrelated property-value pairs and cascading imprecision. Our technique regains precision by tracking such correlated read/write pairs and analyzing them separately for each value of `p`. The analysis thus ends up only

copying values between properties of the same name, dramatically improving precision *and* performance in many cases. This *correlation tracking* is achieved by extracting the relevant code into new functions and analyzing them with targeted context sensitivity.

We implemented our technique as an extension to the Watson Libraries for Analysis (WALA) [26] and conducted experiments on five widely-used JavaScript frameworks: *dojo*, *jquery*, *mootools*, *prototype.js*, and *yui*. On these benchmarks, WALA's default implementation of a field-sensitive Andersen-style analysis usually is not able to complete analysis within a reasonable amount of time and produces very imprecise results. We show that correlation tracking significantly improves both analysis performance and precision: most benchmarks can now be analyzed in seconds, though some scalability challenges remain.

The presence of `eval` and other constructs for executing dynamically generated code means that a (useful) static analysis for JavaScript cannot be sound, and ours is no exception. Nevertheless, there are interesting applications for unsound call graphs in security analysis and bug finding [12, 24], and we expect existing tools to benefit significantly from our techniques.

The contributions of this paper can be summarized as follows:

- We show that a standard implementation of field-sensitive Andersen's points-to analysis extended to handle dynamic property accesses has $O(N^4)$ worst-case running time, in contrast to the $O(N^3)$ bound for other languages.
- We demonstrate that this increased complexity has practical repercussions by showing that a previously developed field-sensitive implementation of Andersen's points-to analysis for JavaScript is unable to analyze several widely-used JavaScript frameworks.
- We present a technique to address scalability issues caused by dynamic property accesses by enhancing the points-to analysis to track correlated dynamic property accesses that must at run-time refer to properties with the same name.
- We report on an implementation of our correlation tracking technique on top of WALA and its application to JavaScript frameworks, demonstrating significantly improved scalability and precision in many cases.

The remainder of this paper is organized as follows. Section 2 presents a motivating example that illustrates the complexity of points-to analysis for JavaScript. Section 3 formulates field-sensitive points-to analysis for JavaScript and shows the $O(N^4)$ worst-case running time of a standard implementation extended with handling of computed property names. Section 4 presents our approach for improved handling of dynamic property accesses. Experimental results are presented in Section 5. Section 6 discusses how similar scalability issues may arise in other languages, demonstrating that the techniques presented in this paper may be more widely applicable. Finally, related work is discussed in Section 7 and conclusions are presented in Section 8.

```
6  function extend(destination, source) {
7    for (var property in source)
8      destination[property] = source[property];
9    return destination;
10 }
11
12 extend(Object, {
13   extend:         extend,
14   inspect:        inspect,
15   ...
16 });
17
18 Object.extend(String.prototype, (function() {
19   function capitalize() {
20     return this.charAt(0).toUpperCase()
21       + this.substring(1).toLowerCase();
22   }
23   function empty() {
24     return this == '';
25   }
26   ...
27   return {
28     capitalize:     capitalize,
29     empty:          empty,
30     ...
31   };
32 })());
33 "javaScript".capitalize(); // == "Javascript"
```

Fig. 1. The `extend` function and some of its uses in *prototype.js*; the definition of `inspect` is omitted

2 Motivation

In this section, we illustrate how some of JavaScript's dynamic features impact points-to analysis and call-graph construction. We illustrate these points using Figure 1, which shows a few fragments of the widely used *prototype.js* library.[2]

2.1 JavaScript Objects and Functions

JavaScript's model of objects and functions is extremely flexible:

– Unlike class-based languages like Java, JavaScript has no built-in concept of an object instance method. Instead, functions are first-class values, and methods are simply functions stored in object properties. For instance on

[2] http://www.prototypejs.org/

lines 12–16 in Figure 1, an object is created with two properties, `extend` and `inspect`, and `extend` is bound to the function defined on line 6.

- Object properties are dynamic in that they are not declared and can be accessed with first-class names. On line 8, the `destination` object has properties assigned based on the `property` variable; if a given property exists in `destination`, it is overwritten; if not, it is created. Thus, the set of properties an object may have is not evident from the code, unlike in a static language like Java.
- The notion of a method call is idiosyncratic; since functions are assigned dynamically to objects, the notion of a receiver is defined by the call itself. Consider the `extend` function defined on line 6. On line 18, this function will be invoked by the `Object.extend` call, and `this` will be bound to `Object`. However, on line 12, `extend` is called directly by name, and `this` defaults to the global object.

Since there is no a priori distinction between properties containing values and properties containing methods, a *field-sensitive analysis*, which represents each property of each abstract object separately, is necessary for obtaining a precise call graph for JavaScript programs. A field-insensitive analysis, which uses a merged representation for each object's properties, would conclude that the invocation of `Object.extend` could possibly invoke `Object.inspect` as well, a very imprecise result.

Several techniques used in other languages to remove obviously invalid results from points-to sets are not applicable in JavaScript. The language lacks classes and declared types for variables, so type filters [9] cannot be employed. Properties are created upon first write, so assignments to non-existent properties cannot be discarded as invalid by the analysis. Finally, although functions declare formal parameters, they can be invoked with any number of actual arguments. If too few arguments are passed, the remaining parameters are assigned the value `undefined`. All arguments (including those not corresponding to any declared parameter) can be accessed via the built-in `arguments` array. This flexibility makes it impossible to perform arity matching to narrow down the set of functions that may be invoked at a given call site.

As a consequence, local imprecision in the analysis can easily cascade and pollute much of the analysis result.

2.2 Dynamic Property Accesses

Figure 1 illustrates how JavaScript's *dynamic property accesses* pose a major challenge for points-to analysis. In particular, the example illustrates how *prototype.js* uses such accesses to dynamically extend objects, a feature often used within *prototype.js* itself. Several other frameworks, including *jquery*, offer similar functionality.

The program of Figure 1 declares a function `extend` on lines 6–10. It uses a `for-in` loop to iterate over the names of all properties of the object bound to its `source` parameter, assigning them to the loop variable `property`.

The value of each property is then read using a *dynamic property access* expression `source[property]`, and stored in a property of the same name in object `destination` by assigning it to `destination[property]`. The following aspects of JavaScript's semantics should be noted:

- The name of the property accessed by a dynamic property access expression is *computed at run-time*.
- As mentioned above, a write to a property *creates* that property if it does not exist yet.

Together, these observations imply that it is possible for a JavaScript program to create objects with an *unbounded* number of properties, which is impossible in statically typed languages such as Java or C#.

Figure 1 also shows two examples of how `extend` is used inside the *prototype.js* library itself.

- On lines 12–16, `extend` is called to bind several functions to properties in the built-in `Object` object. Note that the `extend` function itself is bound to a property `extend` of `Object`.
- On lines 18–32, the `extend` function is invoked as `Object.extend` to extend the `prototype` property of the built-in `String` object with properties `capitalize` and `empty`. As shown on line 33, these properties then become available on all `String` objects, since they have `String.prototype` as their prototype object and hence inherit all its properties.

Now, consider applying Andersen's points-to analysis [3] to the example in Figure 1. As discussed earlier, field-sensitive analysis is necessary to obtain sufficient precision. To handle dynamic property accesses, the analysis must furthermore track the possible values of property-name expressions (like `property` on line 8) and use that information to reason about what properties a dynamic access can read or write. Section 3 formulates such an analysis in more detail and discusses the effect of dynamic property accesses on worst-case complexity.

For the example of Figure 1, variable `property` on line 8 may be bound to the name of any property of any object bound to `source`. In particular, `property` may refer to any property name of the object passed as the second argument in the call on line 12 (`"extend"`, `"inspect"`, etc.) and the one passed on line 18 (`"capitalize"`, `"empty"`, etc.). This means that the points-to set for the dynamic property expression `source[property]` must include *all properties* of the `source` objects. The write to `destination[property]` therefore causes Andersen's analysis to add *all* of these functions to the points-to sets for properties `"extend"`, `"inspect"`, `"capitalize"` etc. in all the `destination` objects (recall that a write to a non-existent property creates the property). In particular, such an analysis would conclude, very imprecisely, that the call `Object.extend(...)` on line 18 might invoke any of the functions `Object` is extended with on line 12.

By the same reasoning, it can be seen that due to the invocation of `extend()` at line 18, this points-to analysis would compute for each property added to `String.prototype` a points-to set that includes `capitalize`, `empty`, and any

```
34 function extend(destination, source) {
35   for (var property in source)
36     (function(p){
37       destination[p] = source[p];
38     })(property);
39   return destination;
40 }
```

Fig. 2. Transformed example

other function stored in the object literal on line 32. Consequently, an invocation of a function read from one of these properties would be approximated as a call to any one of them by the analysis. The resulting loss of precision is detrimental because `String` objects are used pervasively.

This kind of precision loss arose for several widely-used JavaScript frameworks that we attempted to analyze (see Section 5), making straightforward field-sensitive points-to analysis intractable due to the long time it takes to compute the highly imprecise points-to relation, and the excessive space required to store it. This problem is exacerbated by the fact that JavaScript frameworks use mechanisms such as the `extend` function of Figure 1 internally during initialization, which means that merely including the code for these libraries in a web page will trigger the problem.

Our Technique. In Section 4, we propose *correlation tracking* as a solution to this problem that can dramatically improve both precision and performance. The key idea is to enhance Andersen's analysis to track correlations between dynamic property reads and writes that use the same property name. For our example, the value read from `source[property]` is written into `destination[property]`; since the value of `property` cannot have changed between the read and the write, the enhanced analysis can reason that a property value from `source` may only be copied to the property with the same name in `destination`.

This is implemented by first extracting the relevant code—in this case the body of the `for-in` loop—into a new function with the property name as its only parameter, as shown for function `extend` in Figure 2.[3] The new function is then analyzed context-sensitively with a separate context for each value of the property name parameter, thereby achieving the desired precision.

This context-sensitivity policy is reminiscent of Agesen's Cartesian Product Algorithm [1] and object-sensitive analyses [18, 20] in the sense that different contexts are introduced for a function based on the values passed as arguments (further discussion in Section 4.2). These enhancements enable our analysis to efficiently compute call graphs for framework-based JavaScript applications in many cases that could not be handled by the baseline field-sensitive analysis.

[3] Other variables of the surrounding scope remain accessible in the extracted code, since JavaScript supports lexical scoping.

Table 1. Our formulation of field-sensitive Andersen's points-to analysis in the presence of first-class fields

Statement	Constraint	
`x = {}`^{i}	$\{o^i\} \subseteq pt(x)$	[Alloc]
`v = "name"`	$\{\texttt{name}\} \subseteq pt(v)$	[StrConst]
`x = y`	$pt(y) \subseteq pt(x)$	[Assign]
`x[v] = y`	$\dfrac{o \in pt(x) \qquad \texttt{s} \in pt(v)}{pt(y) \subseteq pt(o.\texttt{s})}$	[StoreField]
`y = x[v]`	$\dfrac{o \in pt(x) \qquad \texttt{s} \in pt(v)}{pt(o.\texttt{s}) \subseteq pt(y)}$	[LoadField]
`v = x.nextProp()`	$\dfrac{o \in pt(x) \qquad o.\texttt{s} \text{ exists}}{\{\texttt{s}\} \subseteq pt(v)}$	[PropIter]

3 Field-Sensitive Points-To Analysis for JavaScript

In this section, we formulate a field-sensitive points-to analysis for a core language based on the object model of JavaScript. This formulation describes the existing points-to analysis implementation in WALA [26], which we use as our baseline. Then, we show that a standard implementation of Andersen's analysis runs in worst-case $O(N^4)$ time for this formulation, where N is the size of the program, due to computed property names. Finally, we give a minimal example illustrating the imprecision that our techniques address.

Formulation. The relevant core language features of JavaScript are shown in the leftmost column of Table 1. Note that property stores and loads act much like array stores and loads in a language like Java, where the equivalent of array indices are string constants.[4] Property names are first class, so they can be copied between variables and stored and retrieved from data structures. As discussed in Section 2, properties are added to objects when values are first stored in them. The `v = x.nextProp()` statement type is used to model the JavaScript `for-in` construct (see Section 2); it updates `v` with the next property name of the object `x` points to.[5] So, assuming a corresponding `hasNextProp` construct, `for (v in x) { B }` could be modeled as:

[4] In full JavaScript, not all string values originate from constants in the program text; as discussed further in Section 5.1, we handle this by introducing a special "unknown" property name that is assumed to alias all other property names.

[5] Property names from objects in the prototype chain are also considered [7, §12.6.4], but we elide this detail here for clarity.

```
while (x.hasNextProp()) { v = x.nextProp(); B }
```

The second column of Table 1 presents Andersen-style points-to analysis rules for the core language. The only way in which this differs from a standard Andersen-style analysis for Java [21] is that it supports tracking of property names as they flow through assignments. We represent the points-to set of a program variable x as $pt(x)$. The rules are presented as inclusion constraints over points-to sets of program variables and of properties of abstract objects (e.g., o.`name`). We assume that object allocations are named with one abstract heap object per static statement, e.g., abstract object o^i for statement i. Note that pt-sets track not just abstract objects, but also string constants possibly representing property names.[6]

Complexity. Computing an Andersen-style points-to analysis can be viewed as solving a *dynamic transitive closure* (DTC) problem for a graph of constraints similar to those in Table 1: $o \in pt(x)$ iff x is reachable from o in the graph. Reachability information is stored by maintaining points-to sets for variables and for fields of abstract-locations, and "propagating" abstract locations to points-to sets based on the constraint edges [21]. The problem requires *dynamic* transitive closure since the StoreField and LoadField rules introduce new constraints based on other points-to facts, which translates to adding new graph edges based on other reachability facts. Most efficient implementations of Andersen's analysis essentially work by computing a dynamic transitive closure; see previous work for details [21].

For Java-like languages, the worst-case complexity of the DTC computation for points-to analysis is $O(N^3)$. The key constraint rules to consider are for field accesses, e.g., the StoreField rule for a statement `x.f = y` (reasoning about LoadField is similar):

$$\frac{o \in pt(x)}{pt(y) \subseteq pt(o.\texttt{f})}$$

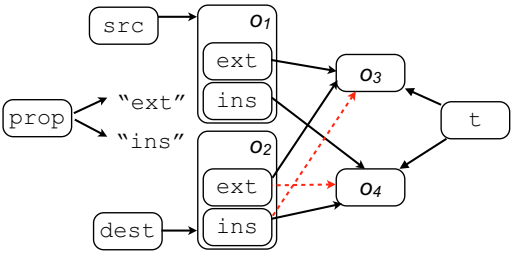
Note that since the field name is manifest in the Java statement, the field-name precondition seen in Table 1 is not required in this rule. Via this rule, the algorithm may add $O(N)$ constraints of the form $pt(y) \subseteq pt(o.\texttt{f})$ to the graph in the worst case (since $|pt(x)|$ is $O(N)$). Considering $O(N)$ abstract locations that may be propagated across each such generated constraint, and $O(N)$ field-write statements in the program, we obtain an $O(N^3)$ worst-case bound on running time.

Now, consider the StoreField rule from Table 1, which includes an additional pre-condition $\texttt{s} \in pt(v)$ to handle computed property names. Unlike Java, this rule may introduce $O(N^2)$ new constraints, one for each (abstract location, property name) pair. Factoring in $O(N)$ worst-case propagation work for each constraint and $O(N)$ store statements in the program now yields an $O(N^4)$ running-time bound for the analysis, worse than that for Java. This bound assumes that the analysis may find each abstract location to have $O(N)$ fields in

[6] If a non-`String` object o is used as a property name in a dynamic property access, a name is obtained by coercing o to a `String` [7, §11.2.1]; we elide modeling of this behavior here for clarity.

```
1 src = {}
2 dest = {}
3 src["ext"] = {}
4 src["ins"] = {}
5 prop = (*) ? "ext" : "ins";
6 t = src[prop];
7 dest[prop] = t;
```

(a)



(b)

Fig. 3. Imprecisely analyzed property accesses, and the corresponding points-to graph computed by the analysis in Table 1; the red, dashed edges are spurious

the worst-case. In a language with classes, an assumption of a constant number of fields per object becomes reasonable [21]. But, with JavaScript's lack of classes and semantics of creating fields when written, such an assumption cannot be made. In fact, our techniques are designed to address a common pattern that causes a blowup in the number of fields per object, as discussed below.

$$\dfrac{\dfrac{\dfrac{o_1 \in pt(src)}{o_4 \in pt(o_1.\texttt{ins})}\,\text{L4} \qquad \dfrac{o_1 \in pt(src) \qquad \texttt{ins} \in pt(prop)}{pt(o_1.\texttt{ins}) \subseteq pt(t)}\,\text{L6}}{o_4 \in pt(t)} \qquad \dfrac{\texttt{ext} \in pt(prop) \qquad o_2 \in pt(dest)}{pt(t) \subseteq pt(o_2.\texttt{ext})}\,\text{L7}}{o_4 \in pt(o_2.\texttt{ext})}$$

Fig. 4. Imprecise derivation of $o_4 \in pt(o_2.\texttt{ext})$ for the example of Figure 3(a), using the rules of Table 1. Rule applications are labeled with the corresponding line number from Figure 3(a) as appropriate.

Imprecision Example. Consider the sequence of statements in Figure 3(a). This program is intended to model normalized statements corresponding to the `for-in` loop in Figure 1 as they would appear to Andersen's analysis; identifier names have been abbreviated.[7] The program creates properties `ext` and `ins` in the object o_1 (named by the line number of its allocation), and then copies one of these properties to object o_2. Figure 3(b), which gives the points-to relation computed for the program, shows that the analysis has imprecisely conflated the `ext` and `ins` properties in o_2, concluding that they both may point to either o_3 or o_4.

Figure 4 shows in detail how the analysis imprecisely concludes that $o_4 \in pt(o_2.\texttt{ext})$, based on the rules of Table 1. Here, the imprecision stems from failing to take into account that lines 6 and 7 of Figure 3(a) must read the same property name from variable *prop*—the rule applied for line 6 uses $\texttt{ins} \in pt(prop)$, while the line 7 rule uses $\texttt{ext} \in pt(prop)$.

Note that conversion of JavaScript statements to their normal form for Andersen's analysis (see Table 1) may in fact *introduce* the imprecision illustrated above. Say that lines 6 and 7 of Figure 3(a) were written as a single statement `dest[prop] = src[prop]`. A points-to analysis that processed this statement directly

[7] We avoid full normalization to the statement types of Table 1 to ease readability.

```
1  src = {}
2  dest = {}
3  src["ext"] = {}
4  src["ins"] = {}
5  if (*) {
6    prop1 = "ext";
7    t1 = src[prop1];
8    dest[prop1] = t1;
9  } else {
10   prop2 = "ins";
11   t2 = src[prop2];
12   dest[prop2] = t2;
13 }
```

(a)

```
1 src = {}
2 dest = {}
3 src["ext"] = {}
4 src["ins"] = {}
5 prop = (*) ? "ext" : "ins";
6 (function(ff) {
7   t = src[ff];
8   dest[ff] = t;
9 })(prop);
```

(b)

Fig. 5. Transformed versions of the example of Figure 3, to illustrate our technique

could avoid the above imprecision—even without flow sensitivity, it is clear that `prop` cannot be re-defined between its two uses in this statement.[8] However, a flow-insensitive analysis must allow for `prop` to be re-defined between the normalized statements `t = src[prop]; dest[prop] = t`, adding imprecision. In contrast to a previous study showing that this normalization does not cause imprecision for points-to analysis of C in practice [5], we have observed real examples (e.g., that of Figure 1) where normalization causes imprecision for JavaScript. Our techniques can recover precision in these cases, and also in cases where multiple source-level statements are relevant, as discussed in the next section.

4 Correlation Tracking

We now discuss our *correlation tracking* technique for improving the scalability of JavaScript points-to analysis in practice. We first illustrate the technique on a small example (Section 4.1), and then detail how we achieve the correlation tracking by extracting code into new functions and analyzing them with targeted context sensitivity (Section 4.2).

4.1 Example

Recall that in the example from Figure 3 of Section 3 the field-sensitive points-to analysis of Table 1 imprecisely concluded that $o_4 \in pt(o_2.\texttt{ext})$ since it did not track the fact that the property read on line 6 must refer to the same property name as the write on line 7.

We can force the analysis to recognize this by splitting variable `prop`, which contains the property name, into two variables `prop1` and `prop2`, respectively corresponding to `prop`'s possible values of `"ext"` and `"ins"`, as shown in Figure 5(a).

[8] Unfortunately, no polynomial-time algorithm is known for such an analysis [5].

Considering again the derivation from Figure 4, we see that the analysis can derive a modified constraint $pt(t_1) \subseteq pt(o_2.\texttt{ext})$ for Figure 5(a) (via line 8), but it cannot derive the corresponding $o_4 \in pt(t_1)$ fact required to imprecisely conclude that $o_4 \in pt(o_2.\texttt{ext})$. Instead, the analysis can only derive $o_4 \in pt(t_2)$ (via line 11), leading to $o_4 \in pt(o_2.\texttt{ins})$ (line 12), which is in fact feasible.

This example in Figure 5(a) is handled more precisely since the cloning enables the points-to analysis to *track the correlation* of the property name between the copied dynamic property reads and writes—it only copies `src["ext"]` to `dest["ext"]` and `src["ins"]` to `dest["ins"]`.

In general, of course, it is not straightforward to determine the set of possible property names a correlated read/write pair may refer to. Thus, instead of cloning the relevant section of code once for every property name, we extract it into a fresh anonymous function taking the property name as a parameter as shown in Figure 5(b).[9] Combined with a special context sensitivity policy that analyses the fresh function separately for every (abstract) parameter value, we obtain the same precision as with cloning.

4.2 Implementing Correlation Tracking

Identifying Correlations. A dynamic property read r and a property write w are said to be *correlated* if w writes the value read at r, and both w and r must refer to the same property name.

We identify such correlated read/write pairs by a data flow analysis on an intra-procedural def-use graph. Starting from a dynamic property read r of the form `o[p]`, we follow def-use edges to track where the read value flows. If it may flow into a dynamic property write w of the form `o'[p'] = e` and we can prove that `p` must have the same value as `p'`, then r and w are correlated. In practice, we have found it sufficient to only consider cases where `p` and `p'` refer to the same local variable p, and p is not redefined between r and w.

In some cases, r and w may be in different functions. To capture this situation, we conservatively assume that any called function may perform a dynamic property write. Thus, if both the value read by r and the value of `p` flow into a function call c, we consider r and c to be correlated. The reverse situation, with the property read occurring in a callee, does not appear to occur in practice and we do not handle it.

While this analysis is not complete and hence will not identify all correlated pairs, it is simple to implement and suffices in practice.

Function Extraction. Once we have identified a correlation between r and w on property name p, we extract the snippet of code containing the two accesses into a new function with p as its parameter. In practice, it turns out to be sufficient to only consider the case where the read r and the write w (or the call c for inter-procedural correlations) occur in two statements s_r and s_w such that s_r precedes s_w inside the same block of statements.

[9] Note that local variables `src` and `dest` are accessible inside the anonymous function due to JavaScript's lexical scoping discipline.

If the extraction regions corresponding to different correlated pairs overlap, they are merged and extracted into a function taking all the relevant property name parameters as arguments.

Some language constructs need to be treated specially. In particular, the `this` value of the enclosing method needs to be passed as a separate parameter to the extracted function if there are any `this` references in the extraction region, and these references must be rewritten to access the parameter instead. We currently do not extract code that references the `arguments` array. Finally, unstructured control flow (such as `continue` or `break`) across the boundaries of the extraction region needs to be rewritten to instead return a special flag value; this value is checked by the enclosing function, and the appropriate jump is performed. All these checks and transformations are of the same kind as those performed by implementations of the Extract Method refactoring [19].

Context Sensitivity. To ensure that correlated pairs are analyzed once per property name, we analyze the extracted function using the following context sensitivity policy:

> If function f uses a parameter p as the property name in a dynamic property access, f is analyzed context-sensitively, with a separate context for each value of p.

For the example of Figure 5(b), the policy creates a separate context for each value of the `ff` parameter for the function at line 6. This policy effectively clones the extracted function for each possible value of `ff` (`"ext"` and `"ins"`), matching the cloning in Figure 5(a) and hence adding the desired correlation tracking. Our context-sensitivity policy can be viewed as a variant of object sensitivity [18, 20], using the property name parameter instead of the `this` parameter to distinguish contexts.[10]

The policy is not restricted to functions generated by the extraction of correlated pairs. Thus it is able to handle cases where correlated reads and writes happen in different functions. For instance, consider the following example (loosely based on code found in the *mootools* framework):

```
function doWrite(d,p,v) { d[p] = v; }
function extend(destination, source) {
  for (property in source)
    doWrite(destination,property,source[property]);
}
```

Here, the read in `extend` is correlated with the write in `doWrite`. Our intraprocedural analysis will identify this correlation due to its conservative treatment of function calls, hence the call to `doWrite` will be extracted into a fresh function with parameter `property`. Both the fresh function and `doWrite` use their

[10] In the case where a function uses multiple parameters as property names in dynamic accesses, we choose one such parameter arbitrarily to distinguish contexts. We have not observed this case in practice.

parameter as a property name, hence they are both analyzed context sensitively, yielding the desired precision.

This additional context sensitivity does not improve the worst-case running time of the analysis; in fact, the analysis could in principle become slower since more constraints are generated for functions analyzed under the new contexts. As the next section shows, however, the technique dramatically improves scalability in practice because we end up creating much sparser points-to graphs.

5 Evaluation

Here we present an experimental evaluation of the effectiveness of our techniques to make field-sensitive points-to analysis for JavaScript scale in practice.

5.1 Implementation

Our analysis implementation is built on top of WALA [26]. WALA provides a points-to analysis implementation for JavaScript, which we extended with correlation tracking. WALA's JavaScript points-to analysis is built on a highly-tuned constraint solver also used for Java points-to analysis [21], and it has already been used in production-quality security analyses for JavaScript [12, 24]. Our work was motivated by the fact that WALA's analysis could not scale to analyze many JavaScript frameworks. By building on WALA, we were able to re-use its handling of various intricate JavaScript language constructs such as the prototype chain and `arguments` array (also discussed in previous work [10, 14]). WALA also provides handwritten models of various pre-defined JavaScript objects and standard library functions.

Default Context Sensitivity. WALA's JavaScript points-to analysis uses context sensitivity by default to handle two key JavaScript language features, and we preserved these techniques in our modified version of the analysis. The first construct is `new`, used to allocate objects. The `new` construct has a complex semantics in JavaScript based on dispatch to a first-class function value [7, §11.2.2].[11] WALA handles `new` by generating synthetic functions to model the behaviors of possible callees. As any one of these synthetic functions may be invoked for multiple `new` expressions, they must be analyzed with one level of call-string context in order to achieve the standard allocation-site-based heap abstraction of Andersen's analysis.

Accesses to variables in enclosing lexical scopes are also handled via context sensitivity by WALA. Handling lexical scoping for JavaScript can be complicated, as nested functions may read and/or write variables declared in enclosing functions [7, §10.2]. WALA employs contexts to ensure that a lexical access only reads or writes the appropriate clones of the accessed variable in the call graph. Without this approach, lexical accesses may lead to merging across call graph

[11] In some cases, a `new` expression may not even create an object [7, §15.2.2.1].

clones, muting the benefits of other context-sensitivity policies. Also, lexical information is stored in abstract-location contexts for function objects as needed, to handle closure accesses performed by the function after it is returned from its enclosing lexical scopes. Note that our function extraction technique is eased by WALA's precise treatment of lexical accesses, as fewer parameters and return values need to be introduced (see Section 4.2).

Unknown Properties. While our analysis formulation in Section 3 allowed for only constant strings as property names, in a JavaScript property access `a[e]`, `e` may be an arbitrary expression, computed using user inputs, arithmetic, complex string operations, etc. Hence, in some cases WALA cannot compute a complete set of constant properties that a statement may access, i.e., the statement may access an *unknown* property. WALA handles such cases conservatively via *abstract object properties*, each of which represents the values stored in all properties of some (abstract) object. When created, an abstract property is initialized with all possible property values discovered for the object thus far. A read of an unknown object property is modeled as reading the object's abstract property, while a write to an unknown property is treated as possibly updating the object's abstract property and any other property whose name is known. This strategy avoids pollution in the case where all reads and writes are to known constant property names.

`call` *and* `apply`. JavaScript function objects provide two built-in methods `call` and `apply` to reflectively invoke the represented function [7, §15.3.4]. Whenever the analysis determines that a call may dispatch to one of these methods, it creates a synthetic stub function that models the reflective call taking place at this call site.

Soundness. WALA's points-to analysis attempts to treat most commonly used JavaScript constructs conservatively. However, unsoundness will occur in some cases:

- We currently do not handle `with` blocks, which put the properties of an object in the local scope. Of the frameworks we evaluate, only one (*dojo*) uses `with` blocks in two places. We manually desugared these uses in a similar way as suggested in [13].
- We do not model the semantics of `eval` and the `Function` constructor as well as several other constructs for executing dynamically generated code. This is analogous to how analyses of Java commonly ignore complex reflection and dynamic code loading.
- Some implicit conversions prescribed by the language standard are not yet modeled. In particular, some of these conversions can result in calls to `toString` or `valueOf` methods that we currently ignore.
- Our model of the JavaScript library and the DOM is incomplete, which can lead to unsoundness. Again, this is similar to how analyses of Java work, few of which model the intricate native implementation of portions of the libraries.

Table 2. Overview of the JavaScript frameworks used in our experiments. The "LOC" column gives the number of lines of non-blank, non-comment source code as determined by CLOC (`http://cloc.sf.net`), and the "Correlated Pairs" column gives the number of correlated access pairs extracted by our technique.

Framework	Home Page	Version	LOC	Correlated Pairs
dojo	`http://www.dojotoolkit.org`	1.6.1	4748	20
jquery	`http://jquery.com`	1.6.1	5896	34
mootools	`http://mootools.net`	1.4.0	3815	41
prototype.js	`http://prototypejs.org`	1.7	4956	9
yui	`http://developer.yahoo.com/yui`	2.9	24088	31

In spite of possible unsoundness, the points-to analysis is still useful for a variety of clients, e.g., bug-finding tools [12, 24]. Furthermore, we expect that correlation tracking would provide significant benefits for a sound approach to JavaScript points-to analysis as well.

5.2 Experimental Setup

We evaluated our approach on five popular JavaScript frameworks listed in Table 2, which are among the most widely used frameworks according to a recent survey [27]. For each framework, we collected six small benchmark applications that use the framework, ranging from trivial web pages that do nothing but load the framework scripts to toy web applications of up to 155 lines of code.[12] Note that even just loading each framework already causes significant initialization code to run.

For each benchmark, we attempted to construct call graphs (and hence points-to graphs) using both WALA's standard points-to analysis and our improved technique. We found that most of the frameworks contain sophisticated uses of the above-mentioned reflective methods `call` and `apply`. To more clearly separate out the impact of these features (which is orthogonal to the issue addressed by correlation tracking) we additionally ran our analysis once with modeling of `call` and `apply`, and once without, thus yielding a total of four different configurations.

From now on, we will refer to the configuration using WALA's standard analysis without `call`/`apply` support as "Baseline$^-$" and to the one with support as "Baseline$^+$"; "Correlations$^-$" and "Correlations$^+$" are the corresponding configurations using correlation tracking.

Table 2 also shows, in its last column, the number of correlated access pairs that our technique extracts into fresh functions, which is relatively modest. The benchmark applications themselves did not contain any correlated access pairs.

We performed a separate manual transformation of the `extend` function in *jquery* to eliminate its complex use of the `arguments` array, which again is orthogonal to our focus in this paper. Here is an excerpt of the relevant code:

[12] A complete list of the benchmark applications used and all of our experimental data is available online at `http://tinyurl.com/JSPointers`.

```
jQuery.extend = function () {
  var target = arguments[0] || {}, i = 1,
      length = arguments.length, deep = false;
  // Handle a deep copy situation
  if ( typeof target === "boolean" ) {
    deep = target;
    target = arguments[1] || {};
    // skip the boolean and the target
    i = 2;
  }
  ...
  // extend jQuery itself if only one argument is passed
  if ( length === i ) {
    target = this;
    --i;
  } ...
}
```

The function explicitly tests both the number of arguments and their types, with significantly different behaviors based on the results. If the first argument is a boolean, its value determines whether a deep copy is performed, and if there is only one argument, then its properties are copied to `this`. Any sort of traditional flow-insensitive analysis of this function gets hopelessly confused about what is being copied where, since `target`, the destination of the copy, can be an argument, a fresh object, or `this` depending upon what is passed.

We manually specialized the above function for the different possible numbers and types of arguments, and this specialized version is analyzed in all four configurations of the points-to analysis. Without the specialization, neither the baseline analysis nor our modified version is able to build a call graph for `jquery`. We leave it to future work to build an analysis to automatically perform these specializations.

All our experiments were run on a Lenovo ThinkPad W520 with a 2.20 GHz Intel Core i7-2720QM processor and 8GB RAM running Linux 2.6.32. We used the OpenJDK 64-Bit Server VM, version 1.6.0_20, with a 5GB maximum heap.

5.3 Results

Performance. We first measured the time it takes to generate call graphs for our benchmarks using the different configurations, with a timeout of ten minutes. The results are shown in Table 3. Since our benchmarks are relatively small, call graph construction time is dominated by the underlying framework, and different benchmarks for the same framework generally take about the same time to analyze. For this reason, we present average numbers per framework, except in the case of *dojo* where one benchmark took significantly longer than the others; its analysis time is not included in the average and given separately in parentheses.

Configuration Baseline$^-$ does not complete within the timeout on any benchmark except for *mootools*, which it analyzes in less than a second on average.

Table 3. Time (in seconds) to build call graphs for the benchmarks, averaged per framework; '*' indicates timeout. For *dojo*, one benchmark takes significantly longer than the others, and is hence listed separately in parentheses.

Framework	Baseline$^-$	Baseline$^+$	Correlations$^-$	Correlations$^+$
dojo	* (*)	* (*)	3.1 (30.4)	6.7 (*)
jquery	*	*	78.5	*
mootools	0.7	*	3.1	*
prototype.js	*	*	4.4	4.5
yui	*	*	2.2	2.1

However, once we move to Baseline$^+$ and take `call` and `apply` into consideration, *mootools* also becomes unanalyzable.

Our improved analysis fares much better. Correlations$^-$ analyzes most benchmarks in less than five seconds, except for one *dojo* benchmark taking half a minute, and the six *jquery* benchmarks, which take up to 80 seconds. Adding support for `call` and `apply` again impacts analysis times: the analysis now times out on the `jquery` and `mootools` tests, along with the `dojo` outlier (most likely due to a sophisticated nested use of `call` and `apply` on the latter), and runs more than twice as slow on the other *dojo* tests; on *prototype.js* and *yui*, on the other hand, there is no noticeable difference. However, our precision measurements indicate that some progress has been made even for the cases with timeouts in Correlations$^+$ (see below).

Our timings for the "+" configurations do not include the overhead for finding and extracting correlated pairs, which is very low: on average, the former takes about 0.1 seconds, and the latter even less than that.

In summary, correlation tracking speeds up the analysis dramatically in most cases, though reflective language features like `call` and `apply` still present a challenge for some of the benchmarks.

Memory Consumption. Introducing a more sophisticated context sensitivity policy may in general lead to larger points-to graphs since the same function may now be analyzed under many more contexts than before, which in turn leads to increased memory consumption. For our benchmarks, we measured the number of points-to edges as a proxy for memory consumption. We found that correlation tracking usually *decreases* this number by two to three orders of magnitude, indicating that the less precise analysis configurations build much denser graphs.

There are two exceptions to this pattern. On *jquery*, the decrease only amounts to 60% to 90%. On *mootools*, there is a case where correlation tracking leads to larger points-to graphs: Correlations$^-$ on average yields about four times as many points-to edges as Baseline$^-$. This is not surprising, since *mootools* is the only benchmark that the imprecise analysis configurations can actually handle.

Reachable Functions. To assess the quality of the generated call graphs, we measured the percentage of functions the analysis considers reachable. In cases

Table 4. Percentage of functions considered reachable by our analysis, averaged by framework; '≥' indicates that the number is a lower bound due to analysis timeout. As before, numbers for the outlier on *dojo* are given separately.

Framework	Baseline$^-$	Baseline$^+$	Correlations$^-$	Correlations$^+$
dojo	≥ 60.8% (≥60.4%)	≥ 60.5% (≥60.1%)	16.7% (24.5%)	18.8% (≥28.3%)
jquery	≥ 35.9%	≥ 36.2%	26.7%	≥ 31.5%
mootools	9.5%	≥ 35.5%	9.5%	≥ 10.9%
prototype.js	≥ 40.5%	≥ 40.7%	17.8%	18.7%
yui	≥ 16.6%	≥ 16.6%	12.0%	12.2%

of timeout, we base our measurements on the partial call graphs available after ten minutes; the numbers then represent lower bounds.

For every framework under every configuration, Table 4 shows the average percentage of reachable functions over all the benchmarks; variance between benchmarks was very low except for the same *dojo* benchmark that produced atypical timings, which is again listed separately. The baseline configurations consistently deem more functions to be reachable than the correlation tracking configurations, in many cases dramatically so, indicating poor call graph quality.

Polymorphism. As a final measure of call graph quality, Table 5 shows the number of highly polymorphic call sites with more than five targets. Once more, these tend to be very similar for benchmarks based on the same framework, so we average over frameworks, except for the by now well-known outlier on *dojo*.

The correlation-tracking configurations report very few highly polymorphic call sites: the maximum number is 11 such sites on the problematic *dojo* benchmark under configuration Correlations$^+$, and the maximum number of call targets is 22 on some of the *jquery* benchmarks. We inspected several of these sites and found that they involved higher-order functions and callbacks, justifying the higher call graph fanout. The baseline configurations, on the other hand, produce very dense call graphs with many highly imprecisely resolved call sites, some with more than 300 call targets.

Note that even for cases where Correlations$^+$ times out, the number of highly-polymorphic call sites is dramatically reduced compared to Baseline$^+$. This result is an indication that correlation tracking is still helpful in these cases, even

Table 5. Number of highly polymorphic call sites (i.e., call sites with more than five call targets) for the benchmarks, averaged per framework; '≥' indicates that the result is a lower bound due to timeout. The outlier on *dojo* is separated out.

Framework	Baseline$^-$	Baseline$^+$	Correlations$^-$	Correlations$^+$
dojo	≥239.4 (≥240)	≥226.4 (≥225)	0.0 (1)	1.0 (≥11)
jquery	≥244.0	≥249.0	3.0	≥9.0
mootools	0.0	≥29.2	0.0	≥0.0
prototype.js	≥164.5	≥166.0	0.0	0.2
yui	≥29.0	≥34.5	0.0	0.0

though further work on scalability is needed. For clients that do not require a full call graph, the partial call graph computed by Correlations$^+$ would likely be more useful than that of Baseline$^+$ due to its lower density.

In summary, these results clearly show that correlation tracking significantly improves scalability and precision of field-sensitive points-to analysis for a range of JavaScript frameworks.

6 Other Languages

We have shown that correlation tracking improves analysis of several common JavaScript frameworks. But while our work focuses on JavaScript, there are analogs in other languages. Some languages allow writing code equivalent to the `extend` function from *prototype.js*, and most languages provide string-indexed maps that can cause a similar precision loss. We briefly discuss both cases.

Dynamic Property Accesses in Python. Like JavaScript, Python is a highly dynamic scripting language with features for reflective property access: `dir` lists all properties of an object, and `getattr` and `setattr` provide first-class property access. An equivalent of the `extend` function of Figure 1 can easily be written:

```
def extend(a, b):
    for f in dir(b): setattr(a, f, getattr(b, f))
```

This style is less idiomatic and pervasive in Python, however, so the kind of imprecision we see when analyzing JavaScript is less likely to occur in practice.

Imprecision with Maps. Maps are a core data structure in many applications and can cause precision loss in the same manner as dynamic property accesses in JavaScript. Most maps have accessors to get and set entries and to list all keys. Consider an example in Java: for server-side web applications, the `HttpSession` class stores state that persists across multiple user interactions with a server that share a session; the following code, to enhance security, sanitizes all this persistent state:

```
for(String n : session.getAttributeNames())
  session.setAttribute(n, sanitize(session.getAttribute(n)));
```

This is essentially the same pattern as `extend`, and will cause imprecision in modeling session state, unless techniques like correlation tracking are employed.

7 Related Work

We distinguish several threads of related work.

Complexity. Chaudhuri [6] presents an optimization to CFL-reachability / recursive state machine algorithms (which can handle standard field-sensitive points-to analysis [22]) that yields $O(N^3/log(N))$ worst-case running time. We conjecture that similar techniques could shave a logarithmic factor from our $O(N^4)$ bound for points-to analysis in the presence of dynamic property accesses, but devising and analyzing such an algorithm remains as future work.

JavaScript Semantics. Guha et al. [13] present a formalization of a core calculus λ_{JS} for JavaScript, which includes computed property names, prototype chains and other troublesome features, but excludes `eval`. Our implementation is not based on translating JavaScript to λ_{JS}, but even with such an approach the key analysis challenges that we face would remain. A complete formalization of JavaScript (again without `eval`) is described by Maffeis et al. [17], but their semantics is too complex to be useful for reasoning about static analyses.

Argument Sensitivity. The Cartesian product algorithm [1] (CPA) and object sensitivity [18] both inspired our context-sensitivity policy for extracted functions (see Section 4.2). These techniques create contexts based on the concrete types of arguments at call sites, thus allowing analysis of a function to be specialized based on what types of values are being passed to it. CPA does this for all parameters, and object sensitivity applies just to the receiver argument.

Smaragdakis et al. [20] conduct a thorough analysis of object sensitivity, classifying the prior work in terms of how it chooses contexts based on receiver objects. They also introduce type sensitivity in which contexts are distinguished not based on abstract objects themselves, but rather on their types. They show that this is a promising approach for improving the cost/precision balance in analysis, but clearly it depends on having a useful notion of static types, which JavaScript lacks.

Other JavaScript Analyses. JavaScript combines the program analysis challenges of a higher-order functional language with those of a very dynamic scripting language, and considerable work has focused on addressing some of these issues.

- Jensen et al. [14, 15] deal with issues arising from JavaScript's prototype-based inheritance and the use of automatic coercions. They construct a detailed lattice of types and adapt the recency abstraction of Balakrishnan et al. [4] to precisely handle writes to inherited properties in constructors. The JSRefactor refactoring tool for JavaScript [8] also includes a points-to analysis for JavaScript, which is influenced by this line of work.
- Vardoulakis and Shivers [25] introduce CFA2 to tackle the limitations of CFA with respect to the deep nesting of first-class function calls common in higher-order languages. They use a continuation passing style transformation of the code and a summarization scheme based on local state to match deeply-nested calls and returns. The DoctorJS type inference tool[13] is based on CFA2; it uses a special form of context sensitivity to analyze for-in loop bodies once for every abstract value of the loop variable, thus achieving (for this special case) a similar effect to our more general technique.

These techniques mostly address other challenges that arise when analyzing dynamic languages such as JavaScript, and are complementary to our work. There is also much work that focuses on problems that are specific to JavaScript:

[13] `www.doctorjs.org`.

- Zheng et al. [28] present a JavaScript analysis to find data races in code used in asynchronous ways in a Web browser. They analyze JavaScript code that uses several of the popular frameworks that we handle (*jquery*, *prototype.js*, and *yui*); however, they do not actually analyze the framework code, but instead design inference rules with the framework semantics encoded.
- Guarnieri and Livshits's Gatekeeper [10] and Gulfstream [11] tools perform points-to analyses for JavaScript for use in security analysis, with a focus on incremental analysis in the face of dynamically loaded code on Web pages. They treat dynamic property accesses precisely when a single possible property name can be determined statically (by a separate constant propagation pass), and otherwise assume that any property might be referenced. They do not focus on the problems caused by constructs such as `for-in` loops.

To the best of our knowledge, none of these systems is able to analyze JavaScript frameworks.

Dynamic Type Inference for Scripting Languages. An et al. [2] present a dynamic analysis for inferring static types in Ruby,[14] sidestepping many of the challenges a static analysis for Ruby would have to face, which are similar to the issues arising in JavaScript. Despite being dynamic, however, their analysis is sound. Their focus is on type inference, so they do not track some information needed for our analysis, like the values of different string constants. Also, their technique requires test inputs, which are not readily available for JavaScript frameworks.

Field Sensitivity. Tripp et al. [23] present a taint analysis for Java that implements a form of field sensitivity when handling common J2EE idioms.[15] J2EE uses a context structure that is essentially a hash table, and it is usually referenced in practice with constant strings as keys. They employ an abstraction of the semantics of the context object rather than the actual Java code, applying field sensitivity to distinguish different constant keys used in each context.

8 Conclusions

JavaScript is a uniquely challenging language for pointer analysis: ubiquitous use of dynamic property accesses and of idioms for iterating across all of an object's properties together complicate points-to analysis in a fundamental way.

We introduce *correlation tracking*, a technique for targeted context sensitivity to handle situations where values from dynamic property reads flow to dynamic writes of the same property. We show that this technique aids analyzing common JavaScript frameworks, dramatically improving scalability and precision in many cases. We also provide a sense of why analysis is so much more expensive by showing that extending a standard implementation of Andersen's analysis with support for dynamic property accesses increases its worst-case running time from $O(N^3)$ to $O(N^4)$, where N is the size of the program.

[14] http://www.ruby-lang.org/
[15] http://download.oracle.com/javaee

Work remains to further improve points-to analysis for JavaScript; while correlation tracking makes many popular frameworks tractable, there are some that still cannot be fully analyzed. Hence, our future work will focus on finding and solving the further causes of complexity in these frameworks, including better handling of `call` and `apply` and automation of the precise handling of the `arguments` array that was crucial for *jquery* (see Section 5). We also plan to integrate our current analysis with existing tools [12, 24], as we expect correlation tracking to significantly improve their effectiveness on framework-based web sites.

References

1. Agesen, O.: The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. An, D., Chaudhuri, A., Foster, J.S., Hicks, M.: Dynamic Inference of Static Types for Ruby. In: POPL (2011)
3. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU (1994)
4. Balakrishnan, G., Reps, T.: Recency-Abstraction for Heap-Allocated Storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
5. Blackshear, S., Chang, B.-Y.E., Sankaranarayanan, S., Sridharan, M.: The Flow-Insensitive Precision of Andersen's Analysis in Practice. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 60–76. Springer, Heidelberg (2011)
6. Chaudhuri, S.: Subcubic Algorithms for Recursive State Machines. In: POPL (2008)
7. ECMA. ECMAScript Language Specification, 5th edn., ECMA-262 (2009)
8. Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., Tip, F.: Tool-supported Refactoring for JavaScript. In: OOPSLA (2011)
9. Grove, D., Chambers, C.: A Framework for Call Graph Construction Algorithms. TOPLAS 23(6) (2001)
10. Guarnieri, S., Livshits, V.B.: Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In: USENIX Security Symposium (2009)
11. Guarnieri, S., Livshits, V.B.: Gulfstream: Incremental Static Analysis for Streaming JavaScript Applications. In: WebApps (2010)
12. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R.: Saving the World Wide Web from Vulnerable JavaScript. In: ISSTA (2011)
13. Guha, A., Saftoiu, C., Krishnamurthi, S.: The Essence of JavaScript. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010)
14. Jensen, S.H., Møller, A., Thiemann, P.: Type Analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
15. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural Analysis with Lazy Propagation. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 320–339. Springer, Heidelberg (2010)
16. Lhoták, O., Hendren, L.: Scaling Java Points-to Analysis Using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)

17. Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized Object Sensitivity for Points-to Analysis for Java. TOSEM 14(1) (2005)
19. Schäfer, M., Verbaere, M., Ekman, T., de Moor, O.: Stepping Stones over the Refactoring Rubicon. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 369–393. Springer, Heidelberg (2009)
20. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick Your Contexts Well: Understanding Object-sensitivity. In: POPL (2011)
21. Sridharan, M., Fink, S.J.: The Complexity of Andersen's Analysis in Practice. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 205–221. Springer, Heidelberg (2009)
22. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-Driven Points-To Analysis for Java. In: OOPSLA (2005)
23. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: Effective Taint Analysis of Web Applications. In: PLDI (2009)
24. Tripp, O., Weisman, O.: Hybrid Analysis for JavaScript Security Assessment. In: ESEC/FSE 2011, Industrial Track (2011)
25. Vardoulakis, D., Shivers, O.: CFA2: A Context-Free Approach to Control-Flow Analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 570–589. Springer, Heidelberg (2010)
26. Watson, T.J.: Libraries for Analysis (WALA), http://wala.sf.net
27. Web Technology Surveys. Usage of JavaScript libraries for websites, http://w3techs.com/technologies/overview/javascript_library/all (accessed March 30, 2012)
28. Zheng, Y., Bao, T., Zhang, X.: Statically Locating Web Application Bugs Caused by Asynchronous Calls. In: WWW (2011)

Soundness of Object-Oriented Languages with Coinductive Big-Step Semantics*

Davide Ancona

DISI, Università di Genova, Italy
davide@disi.unige.it

Abstract. It is well known that big-step operational semantics are not suitable for proving soundness of type systems, because of their inability to distinguish stuck from non-terminating computations. We show how this problem can be solved by interpreting coinductively the rules for the standard big-step operational semantics of a Java-like language, thus making the claim of soundness more intuitive: whenever a program is well-typed, its coinductive operational semantics returns a value.

Indeed, coinduction allows non-terminating computations to return values; this is proved by showing that the set of proof trees defining the semantic judgment forms a complete metric space when equipped with a proper distance function.

In this way, we are able to prove soundness of a nominal type system w.r.t. the coinductive semantics. Since the coinductive semantics is sound w.r.t. the usual small-step operational semantics, the standard claim of soundness can be easily deduced.

1 Introduction

It is well known that standard big-step operational semantics are less amenable to prove soundness of type systems than small-step semantics; several important motivations for this statement can be found in the literature [12,13], but, basically, the main source of all problems is the inability of big-step operational semantics to distinguish stuck from non-terminating computations.

Besides addressing this problem, our work seeks to find simpler, and easy to be automated, techniques for proving soundness of abstract compilation of object-oriented languages [7,4,6,5], a novel approach which aims to reconcile type analysis and symbolic execution, where programs are compiled into a constraint logic program (CLP), and type analysis corresponds to solving a certain goal w.r.t. the coinductive semantics of CLP.

The idea of using coinduction to allow big-step semantics to capture non-terminating computations is not new (see the conclusion for a brief survey); Leroy

* This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems. I would like to thank Erik Ernst for his useful comments and suggestions; many thanks also to Sophia Drossopoulou, Atsushi Igarashi, Alan Mycroft and Elena Zucca for all their useful suggestions and questions.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 459–483, 2012.

and Grall [13] have investigated coinductive operational semantics in the context of functional programming, with the main aim of elaborating techniques for automatically proving the correctness of compilation. Among several approaches considered by the authors, the simplest one consists in interpreting coinductively the standard rules for the big-step operational semantics of lambda-calculus, and then expressing the soundness claim in a very direct way: if an expression e has type τ, then the coinductive semantics of e yields a value v of type τ. Unfortunately, such a claim fails to hold, as shown by the authors themselves, since there exist well-typed non-terminating expressions for which the coinductive semantics is not defined. This happens because only finite values are considered, whereas the values that should be returned by the coinductive semantics of such counter-example expressions correspond to necessarily infinite limits of sequences of finite (that is, inductively defined) values. More formally, if only finite values are considered, then it is not possible to define a complete metric space over the set of possibly infinite proof trees for the judgment of the coinductive semantics.

Interestingly enough, if the same approach is taken for a Java-like language, and, more importantly, infinite values are considered as well, then the claim of soundness holds when expressed in terms of a coinductive big-step semantics.

Figure 1 provides a road-map to the main defined judgments and proved claims in this paper. Symbols $\vdash$ and $\Vdash$ denote judgments defined inductively and coinductively, respectively.

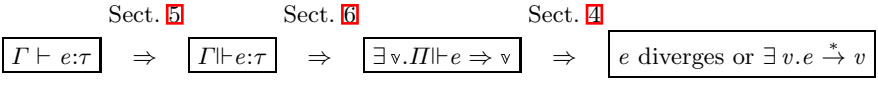


Fig. 1. Relationship between the main judgments

After having introduced the syntax and the standard small-step semantics (abbreviated with ISS) of the language (Section 2), and the mathematical background (Section 3) needed for the proofs, in Section 4 we define the coinductive big-step semantics (abbreviated with CBS) of the language, show, by means of examples, its behavior in case of non-termination, and formalize its relationship with the ISS (rightmost implication in Figure 1): if the CBS of e yields a value $\mathbb{v}$, then the ISS of e either returns a value v, or does not terminate; in other words, whenever the CBS of e yields a value, the ISS of e cannot get stuck, hence, the CBS is sound w.r.t. the ISS. Note the different nature of values (and hence the use of different meta-variables) in the CBS, where they can be infinite, and in the ISS, where they can only be finite.

In Section 5 a conventional inductive and nominal type system is defined (judgment $\Gamma \vdash e{:}\tau$), and a coinductive type system is derived from it (judgment $\Gamma \Vdash e{:}\tau$): such a coinductive system is closer to the CBS, indeed it can be regarded

as an abstraction of the CBS. Furthermore we prove that all judgments that hold in the inductive type system can be derived in the coinductive one as well (leftmost implication in Figure 1).

The core and most difficult part of the formalization concerns the proof of the soundness of the coinductive type system w.r.t. the CBS (middle implication in Figure 1, proved in Section 6). The overview in Figure 1 clearly shows that, as expected, in the end we obtain the the standard soundness result expressed in terms of the ISS. At the end of the section we propose a scheme for proving soundness for a generic type system and language; the rightmost implication in Figure 1 is needed only if one wants to relate the CBS to the ISS, and derive from it a standard soundness claim expressed in terms of the ISS. Furthermore, such an implication needs to be proved just once per programming language, since it does not depend from the considered type system. We expect the definition of the coinductive type system in terms of the inductive one, and the proof of the leftmost implication in Figure 1 to be standard, whereas the proof given in Section 6, with the corresponding definitions of metric spaces, should provide a template to be adapted for proving the middle implication in Figure 1.

Finally, in Section 7 we outline conclusions and related work.

This paper is an extension of a previous work by the same author [3], where the following contributions have been added: The definition of the coinductive type system and the corresponding proof of the leftmost implication in Figure 1 are new; the coinductive type system has been introduced to make the proof of soundness simpler and more modular; it also reveals how coinduction is related to the inductive type system.

While the justification for the introduction of infinite values in the CBS is only informally motivated in the previous work, here it has been made rigorous by means of the notion of complete metric space; in particular, the definition of the metric space of proof trees for the CBS judgment, and the proof of its properties, represent a non trivial task and an original contribution.

In Section 4 a new example has been added (case 2 (c)), showing that soundness does not hold if only infinite but regular values are considered.

All main proofs have been detailed. All omitted proofs and definitions can be found in the companion technical report[1].

2 Definition of the Language

In this section we present our simple Java-like language, which will be used as reference language throughout the paper, together with its standard call-by-value small-step operational semantics.

Syntax: The syntax of the language is defined in Figure 2.

The language is a modest variation of Featherweight Java (FJ) [11], where the main differences concern the introduction of conditional expressions and boolean values, and the omission of type casts.

[1] Available at `ftp://ftp.disi.unige.it/person/AnconaD/ecoop12long.pdf`

$$
\begin{array}{rl}
p & ::= \overline{cd}^{\,n}\ e \\
cd & ::= \textbf{class}\ c_1\ \textbf{extends}\ c_2\ \{\ \overline{fd}^{\,n}\ \overline{md}^{\,k}\ \}\quad (c_1 \neq \texttt{Object}) \\
fd & ::= \tau\ f; \\
md & ::= \tau_0\ m(\overline{\tau\ x}^{\,n})\ \{e\}\quad x_i \neq \textbf{this}\ \forall i = 1..n \\
\tau & ::= c \mid \textbf{bool} \\
e & ::= \textbf{new}\ c(\overline{e}^{\,n}) \mid x \mid e.f \mid e_0.m(\overline{e}^{\,n}) \mid \textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2 \\
 & \quad \mid \textbf{false} \mid \textbf{true}
\end{array}
$$

Assumptions: $n, k \geq 0$, inheritance is acyclic, names of declared classes in a program, methods and fields in a class, and parameters in a method are distinct.

Fig. 2. Syntax of the language

Standard syntactic restrictions are implicitly imposed in the figure. Bars denote sequences of n items, where n is the superscript of the bar and the first index is 1. Sometimes this notation is abused, as in $\overline{f}^{\,h} = \overline{e'}^{\,h}$; which is a shorthand for $f_1 = e'_1; \ldots f_h = e'_h;$.

A program consists of a sequence of class declarations and a main expression. Types can only be class names and the primitive type **bool**; we assume that the language supports boxing conversions, hence *bool* is a subtype[2] of the predefined class `Object`, which is the top type.

A class declaration contains field and method declarations; in contrast with FJ, constructors are not declared, but every class is equipped with an implicit constructor with parameters corresponding to all fields, in the same order as they are inherited and declared. For instance, the classes defined below

```
class P extends Object{bool b; P p;}
class C extends P{C c;}
```

have the following implicit constructors:

```
P(bool b,P p){super(); this.b=b; this.p=p;}
C(bool b,P p,C c){super(b,p); this.c=c;}
```

Method declarations are standard; in the body, the target object can be accessed via the implicit parameter `this`, therefore all explicitly declared formal parameters must be different from `this`. Expressions include instance creation, variables, field selection, method invocation, conditional expressions, and boolean literals.

Small-Step Operational Semantics. The definition of the conventional small-step operational semantics of the language can be found in Figure 3. We follow the approach of FJ, even though for simplicity we have preferred to restrict the semantics to the deterministic call-by-value evaluation strategy.

Values are either the literals **false** or **true**, or object expressions in normal form having shape **new** $c(\overline{v}^{\,n})$. As happens for FJ, the semantics of object

[2] This assumption ensures the existence of the join between types, without introducing union types, to make the typing rule for conditional expressions simpler in the type system defined in Section 5.

creation is more liberal than the expected one; indeed, **new** $c(\overline{v}^n)$ is always a correct expression which reduces to itself in zero steps, even when class c is not declared, or the number of arguments does not match the number of fields of c. As we will see, the big-step semantics follows a less liberal semantics, more in accordance with the standard semantics of mainstream object-oriented languages.

As usual, the reduction relation $\rightarrow$ should be indexed over the collection of all class declarations contained in the program (conventionally called class table), however for brevity we leave implicit such an index in all judgments defined in the paper. The reflexive and transitive closure of $\rightarrow$ is denoted by $\stackrel{*}{\rightarrow}$.

The definition of the standard auxiliary functions *fields* and *meth* is straightforward (see the companion technical report). For compactness, such functions provide semantic and type information at once, since they are instrumental for the definition of both the semantics and the type system of the language. Function *fields* returns the list of all fields which are either inherited or declared in the class, in the standard order and with the corresponding declared types. In the case of the predefined class `Object` the returned list is empty (ϵ); field hiding is not supported, hence *fields* is not defined if a class declares a field with the same name of an inherited one. Function *meth* performs standard method look-up: if $meth(c, m) = \overline{\tau}^n\ \overline{x}^n.e{:}\tau$, then look-up of method m starting from class c returns the corresponding declaration where $\overline{\tau}^n\ \overline{x}^n$ are the formal parameters with their declared types, and e and τ are the body and the declared returned type, respectively. If $meth(c, m)$ is undefined, then it means that look-up of m from c fails.

In rule (fld), if f_i is a field of the class, then the expression reduces to the corresponding value passed to the implicit constructor. If the selected field is not in such a list, then the evaluation of the expression gets stuck.

In rule (inv), if method look-up succeeds starting from the class of the target object, then the corresponding body is executed, where the implicit parameter `this` and the formal parameters are substituted with the target object and the argument values, respectively. The notation $e_i[\overline{x}^n \mapsto \overline{v}^n]$ denotes parallel substitution of the distinct variables $\overline{x}^n$ with values $\overline{v}^n$ in the expression e.

Rules for conditional expressions (ift) and (iff), and for context closure (ctx) are straightforward. Contexts are the standard ones corresponding to left-to-right, call-by-value strategy.

3 Background

In the following, with the term *tree* over a set S we will mean a finitely branching tree with nodes in S that is allowed to contain infinite paths. If t is a tree, we denote with $root(t)$ the root of t.

More rigorously, a tree with infinite paths can be defined in terms of partial functions over finite paths of natural numbers (denoted by $\mathbb{N}^*$) [9,7].

$$v ::= \textbf{new}\ c(\overline{v}^n) \mid \textbf{false} \mid \textbf{true}$$
$$\mathcal{C}[\,] ::= \Box \mid \textbf{new}\ c(\overline{v}^n, \Box, \overline{e}^k) \mid \Box.f \mid \Box.m(\overline{e}^n) \mid v.m(\overline{v}^n, \Box, \overline{e}^k) \mid \textbf{if}\ (\Box)\ e_1\ \textbf{else}\ e_2$$

$$\text{(fld)}\ \frac{\mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n, \quad 1 \le i \le n}{\textbf{new}\ c(\overline{v}^n).f_i \to v_i}$$

$$\text{(inv)}\ \frac{\mathit{meth}(c, m) = \overline{\tau}^n\ \overline{x}^n.e{:}\tau}{\textbf{new}\ c(\overline{v}^k).m(\overline{v'}^n) \to e[\texttt{this} \mapsto \textbf{new}\ c(\overline{v}^k), \overline{x}^n \mapsto \overline{v'}^n]}$$

$$\text{(ift)}\ \frac{}{\textbf{if}\ (\textbf{true})\ e_1\ \textbf{else}\ e_2 \to e_1} \qquad \text{(iff)}\ \frac{}{\textbf{if}\ (\textbf{false})\ e_1\ \textbf{else}\ e_2 \to e_2} \qquad \text{(ctx)}\ \frac{e \to e'}{\mathcal{C}[e] \to \mathcal{C}[e']}$$

Fig. 3. Call-by-value inductive small-step operational semantics

Definition 1. *A tree over a set S is a partial function $t : \mathbb{N}^* \to S$ satisfying the following properties:*

1. *its domain is not empty: $dom(t) \neq \emptyset$;*
2. *its domain is prefix-closed: $p \cdot n \in dom(t)$ implies $p \in dom(t)$, for all $p \in \mathbb{N}^*$, and $n \in \mathbb{N}$;*
3. *if $p \cdot n \in dom(t)$ and $k \le n$, then $p \cdot k \in dom(t)$ for all $p \in \mathbb{N}^*$ and $n, k \in \mathbb{N}$;*
4. *there exists $n \in \mathbb{N}$ s.t. $p \cdot n \notin dom(t)$, for all $p \in \mathbb{N}^*$.*

Every path $p \in dom(t)$ identifies a unique subtree t' of t whose root is $t(p)$: $dom(t') = \{p' \in \mathbb{N}^* \mid p \cdot p' \in dom(t)\}$, and $t'(p') = t(p \cdot p')$ for all $p' \in dom(t')$.

Definition 2. *A* regular *(a.k.a. rational) tree is a tree whose set of subtrees is finite.*

Trivially, every finite tree (that is, tree with only finite paths) is regular, but there exist also infinite trees that are regular.

Definition 3. *A metric space (S, d) is a set S equipped with a function $d{:}S \times S \to \mathbb{R}$, called* metric *or* distance*, satisfying the following properties, for all x, y, and z in S:*

- *(identity) $d(x, y) = 0$ iff $x = y$;*
- *(symmetry) $d(x, y) = d(y, x)$;*
- *(triangle inequality) $d(x, z) \le d(x, y) + d(y, z)$.*

Definition 4. *Let (S, d) be a metric space.*

- *A sequence $(x_i)_{i \in \mathbb{N}}$ has limit l iff for all $\epsilon > 0$ there exists $k \in \mathbb{N}$ s.t. $d(x_n, l) < \epsilon$, for all $n > k$.*
- *A Cauchy sequence $(x_i)_{i \in \mathbb{N}}$ is a sequence s.t. for all $\epsilon > 0$ there exists $k \in \mathbb{N}$ s.t. $d(x_n, x_m) < \epsilon$ for all $n, m > k$.*
- *A metric space is* complete *iff all Cauchy sequences have a limit.*

Proposition 1. *Let T_S be the set of all trees over S. Then, T_S is a complete metric space [8,2] with the following metric:*

$$d_T(t_1, t_2) = 2^{-c}$$

where $c = shtp(t_1, t_2) = \min\{n \in \mathbb{N} \mid p \in \mathbb{N}^n, t_1(p) \neq_\perp t_2(p)\}$, $min\,\emptyset = \infty$, $2^{-\infty} = 0$, $t_1(p) =_\perp t_2(p)$ iff either $p \notin dom(t_1)$ and $p \notin dom(t_2)$, or $p \in dom(t_1) \cap dom(t_2)$ and $t_1(p) = t_2(p)$. That is, c is the length of a shortest path that distinguishes t_1 from t_2, if $t_1 \neq t_2$, or $c = \infty$ if $t_1 = t_2$.

By definition, for all pairs of trees t_1 and t_2, $d_T(t_1, t_2) \in \{0\} \cup \{2^{-c} \mid c \in \mathbb{N}\}$, that is, $0 \leq d_T(t_1, t_2) \leq 1$. It can be proved that the set of finitely branching trees with infinite paths, with the metric defined above, is the (unique up to isometries) completion of the set of finitely branching trees with finite paths with the same metric.

Definition 5. *Let us consider a judgment where all possible instantiations range over the set $\mathcal{J}$.*

A proof tree ∇ *for an instantiation $j \in \mathcal{J}$ of the judgment is a tree ∇ over $\mathcal{J}$ s.t. $root(\nabla) = j$.*

A valid proof tree *for an instantiation $j \in \mathcal{J}$ of the judgment is a proof tree ∇ s.t. for any node j in ∇, if $j_1, \ldots, j_k$ are the children of j, then $\frac{j_1, \ldots, j_k}{j}$ is a correct instantiation of one of the meta-rules defining this kind of judgment.*

We simply write that ∇ is a (valid) proof tree for the judgment, when we are not interested in specifying the particular instantiation $j \in \mathcal{J}$ which is the root of the tree.

We write $ok(\textsc{r})(\nabla)$ to indicate that the root of ∇ together with its children are a correct instantiation of the meta-rule labeled by R. Hence, a valid proof tree ∇ is a proof tree s.t. for all subtrees ∇' of ∇ (including ∇), there exists a meta-rule R s.t. $ok(\textsc{r})(\nabla')$.

Definition 6. *A* complete lattice *is a partially ordered set $(L, \leq)$ s.t. any subset S of L has a supremum (a.k.a. least upper bound) denoted with* $\sup S$.

Since, by definition of inf and sup, $\inf S = \sup\{x \in L | \forall\, y \in S. x \leq y\}$, $\sup \emptyset$ is the least element $\perp$ of L, and $\inf \emptyset$ is the greatest element $\top$ of L, then every subset of a complete lattice has an infimum (greatest lower bound) and every complete lattice is bounded.

Definition 7. *Let $(L, \leq)$ be a complete lattice. A (total) function $f : L \to L$ is* continuous *iff it preserves the supremum of every subset of L: for all $S \subseteq L$, $f(\sup S) = \sup\{f(x) | x \in S\}$.*

It is easy to prove that a continuous function is monotone and preserves infima as well.

Definition 8. *Let $(L, \leq)$ be a partially ordered set, f a (total) function from L to L, and x an element of L.*

- *x is a pre-fixed point of f (a.k.a. f-closed) iff $f(x) \leq x$;*
- *x is a post-fixed point of f (a.k.a. f-dense or f-justified or f-consistent) iff $x \leq f(x)$.*

Trivially, x is a fixed-point of f iff x is both a pre-fixed and a post-fixed point of f.

Theorem 1 (Tarski-Knaster). *Let $(L, \leq)$ be a complete lattice, and $f : L \to L$ a monotone function. Then*

1. *$f(\inf\{x \in L | x$ pre-fixed point of $f\}) = \inf\{x \in L | x$ pre-fixed point of $f\}$;*
2. *$f(\sup\{x \in L | x$ post-fixed point of $f\}) = \sup\{x \in L | x$ pre-fixed point of $f\}$.*

From Theorem 1 one can trivially deduce that a monotone function defined on a complete lattice has always a least fixed-point (which is also the least pre-fixed point), and a greatest fixed point (which is also the greatest post-fixed point).

Given a judgment defined by a set of meta-rules, with instantiations ranging over $\mathcal{J}$, it is possible to define the one step inference function $\mathcal{F}$ over the power-set of $\mathcal{J}$ as follows: for any subset J of $\mathcal{J}$, $\mathcal{F}(J)$ is the subset J' of $\mathcal{J}$ s.t. for any $j \in J'$, there exists a correct instantiation $\frac{j_1, \ldots, j_k}{j}$ of a meta-rule with $\{j_1, \ldots, j_k\} \subseteq J$.

Such a function is always trivially monotone, and it can be proved [7,5,13,17] that its least fixed point is the set of $j \in \mathcal{J}$ s.t. there exists a finite valid proof tree for j, whereas its greatest fixed point is the set of $j \in \mathcal{J}$ s.t. there exists a (possibly infinite) valid proof tree for j.

We denote with $f^n(x)$ n iterated applications of f to x (with $n \in \mathbb{N}$, $f^0(x) = x$).

Theorem 2 (Kleene). *Let $(L, \leq)$ be a complete lattice, and $f : L \to L$ a continuous function. Then*

1. *$\sup\{f^n(\bot) | n \in \mathbb{N}\}$ is the least fixed point of f;*
2. *$\inf\{f^n(\top) | n \in \mathbb{N}\}$ is the greatest fixed point of f.*

Since f is monotone, we have that $f^0(\bot) \leq f^1(\bot) \leq \ldots f^n(\bot) \leq f^{n+1}(\bot) \leq \ldots$ is an ascending chain, and dually, $f^0(\top) \geq f^1(\top) \geq \ldots f^n(\top) \geq f^{n+1}(\top) \geq \ldots$ is a descending chain. Note that the two claims of Theorem 2 hold also under the weaker assumption that f is a monotone function preserving suprema of ascending chains (claim 1), or infima of descending chains (claim 2).

4 A Coinductive Semantics

In this section we define a call-by-value coinductive big-step operational semantics for our language.

Such a semantics is obtained by simply interpreting coinductively the definition of values and the rules of the standard inductive big-step operational semantics (with no rules for error handling).

Definition of the Semantics. The CBS judgment uses value environments (see below), just for uniformity with the type judgment defined in Section 5. Value environments are not strictly necessary, since the rule for method invocation can be equivalently defined with parallel substitution as in ISS. Values are separated from expressions since they are infinite, while expressions are always finite. Such a separation is further stressed by the fact that values belong to a different syntactic category, that is, even finite values are different from expressions.

$$\mathbb{v}, \mathbb{u} ::= obj(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n]) \mid false \mid true \quad (coinductive\ def.)$$

We recall that **false** and **true** are expressions of our language, and values (denoted by the meta-variable v) in the ISS, whereas *false* and *true* are not expressions, but just the corresponding values (denoted by the meta-variable $\mathbb{v}$) in the CBS. Similarly, **new** c(**true**) is both an expression and a value in ISS, whereas $obj(c, [f \mapsto true])$ is the corresponding value in CBS (assuming that the only field of c is f), and is not an expression.

As an example of an infinite value, let us consider the object value $\mathbb{v}$ defined by the equation

$$\mathbb{v} = obj(List, [hd \mapsto obj(Elem, [\,]), tl \mapsto \mathbb{v}])$$

which represents an infinite list; in our language, such a value can only be returned by an infinite computation. Of course in a lazy or imperative language, this value could be returned also by a terminating computation; however, the important point here is that type correct expressions which do not terminate must always return a value in the CBS: as explained in case 2 in the second part of this section, without infinite values the claim of soundness proved in Section 5 would not hold.

The CBS is defined in Figure 4. Thicker lines manifest that rules are interpreted coinductively. A value environment Π is a finite sequence $\overline{x_i}^n \mapsto \overline{\mathbb{v}}^n$, where all variables $\overline{x_i}^n$ are distinct, denoting a finite partial function mapping variables to values ($\emptyset$ denotes the empty environment, $dom(\Pi)$ the domain of Π). Environments model stack frames of method invocations.

Rules (VAR), (FAL), and (TRU) are straightforward. Evaluation of instance creation (NEW) succeeds only if $fields(c)$ is defined (that is, if c and its ancestors are declared in the program and no field is hidden), and returns a list of fields whose length must coincide with the number of arguments; then all arguments are evaluated and the obtained values are associated with the corresponding fields in the object value. For field selection (FLD) the target expression is evaluated; then evaluation succeeds only if an object value is returned, and the selected field is present in the object value; in this case the corresponding associated value is returned. For method invocation (INV) all expressions denoting the target object and the arguments are evaluated. If the value corresponding to the target is an object of class c, method look-up starting from c succeeds and returns a method declaration with a number of formal parameters coinciding with the number of passed arguments, then the method body is evaluated in the environment where

`this` and the formal parameters are associated with their corresponding values. If such an evaluation succeeds, then the returned value is the value of the method invocation. Finally, rules (IFT) and (IFF) deal with the straightforward evaluation of conditional expressions.

$$(\text{VAR})\ \frac{\Pi(x) = \mathbb{v}}{\Pi \Vdash x \Rightarrow \mathbb{v}} \qquad (\text{FAL})\ \frac{}{\Pi \Vdash \mathbf{false} \Rightarrow \mathit{false}} \qquad (\text{TRU})\ \frac{}{\Pi \Vdash \mathbf{true} \Rightarrow \mathit{true}}$$

$$(\text{NEW})\ \frac{\forall\, i = 1..n. \Pi \Vdash e_i \Rightarrow \mathbb{v}_i \quad \mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n}{\Pi \Vdash \mathbf{new}\ c(\overline{e}^n) \Rightarrow \mathit{obj}(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n])} \qquad (\text{IFT})\ \frac{\Pi \Vdash e \Rightarrow \mathit{true} \qquad \Pi \Vdash e_1 \Rightarrow \mathbb{v}}{\Pi \Vdash \mathbf{if}\ (e)\ e_1\ \mathbf{else}\ e_2 \Rightarrow \mathbb{v}}$$

$$(\text{IFF})\ \frac{\Pi \Vdash e \Rightarrow \mathit{false} \qquad \Pi \Vdash e_2 \Rightarrow \mathbb{v}}{\Pi \Vdash \mathbf{if}\ (e)\ e_1\ \mathbf{else}\ e_2 \Rightarrow \mathbb{v}} \qquad (\text{FLD})\ \frac{\Pi \Vdash e \Rightarrow \mathit{obj}(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n]) \quad 1 \le i \le n}{\Pi \Vdash e.f_i \Rightarrow \mathbb{v}_i}$$

$$(\text{INV})\ \frac{\begin{array}{l}\forall\, i = 0..n. \Pi \Vdash e_i \Rightarrow \mathbb{v}_i \quad \mathtt{this} \mapsto \mathbb{v}_0, \overline{x}^n \mapsto \overline{\mathbb{v}}^n \Vdash e \Rightarrow \mathbb{v} \\ \mathbb{v}_0 = \mathit{obj}(c, [\ldots]) \quad \mathit{meth}(c, m) = \overline{\tau}^n\ \overline{x}^n.e{:}\tau\end{array}}{\Pi \Vdash e_0.m(\overline{e}^n) \Rightarrow \mathbb{v}}$$

Fig. 4. Call-by-value coinductive big-step operational semantics

Note that in the CBS, object creation is less liberal than in the ISS: as an example, **new** `c()` is a value in the ISS, whereas the same expression may not evaluate to a value in the CBS; this happens if either c is not declared in the program, or if c contains at least one field.

Coinductive Semantics of Non-terminating Expressions. We have already observed that if the definition of values and the evaluation rules are interpreted inductively, then we obtain a standard inductive big-step operational semantics. Obviously, if an expression evaluates to a value in the inductive semantics, then the same value is obtained in the coinductive one; however, this case concerns terminating expressions, whereas what we do really care about here is the behavior of the CBS for non-terminating expressions. We show that three different cases may occur. All expressions e considered in the examples below are well-typed and do not terminate in the ISS, that is, there exists no normal form e' s.t. $e \xrightarrow{*} e'$.

Case 1: There exist many values $\mathbb{v}$ s.t. $\emptyset \Vdash e \Rightarrow \mathbb{v}$

Let us consider the expression $e =$ **new** `C().m()`, where `C` is the only class declared in the program:

```
class C extends Object{bool m(){this.m()}}
```

Then $\emptyset \Vdash e \Rightarrow \mathbb{v}$ for all values $\mathbb{v}$, as shown in the valid proof tree of Figure 5. Ellipsis means that such a tree is infinite (hence, it cannot be a valid proof for an inductive system), although regular, that is, it can be folded into a finite graph, because of the repeated finite pattern originated from the judgment

$$\dfrac{\emptyset \Vdash \texttt{new C()} \Rightarrow \mathbb{u} \qquad \dfrac{\dfrac{}{\Pi \Vdash \texttt{this} \Rightarrow \mathbb{u}} \qquad \dfrac{\vdots}{\Pi \Vdash \texttt{this.m()} \Rightarrow \mathbb{v}}}{\Pi \Vdash \texttt{this.m()} \Rightarrow \mathbb{v}}}{\emptyset \Vdash \texttt{new C().m()} \Rightarrow \mathbb{v}}$$

Fig. 5. Proof tree for $\emptyset \Vdash e \Rightarrow \mathbb{v}$, where $\mathbb{u} = obj(\texttt{C}, [\,])$, $\Pi = \texttt{this} \mapsto \mathbb{u}$

$\Pi \Vdash \texttt{this.m()} \Rightarrow \mathbb{v}$. Such non-determinism is naturally reflected in the conventional nominal type system (see Section 5) where the return type **bool** can in fact be correctly replaced by any other type defined in the program.

There are also cases where finitely many values are returned. For instance,

$$\emptyset \Vdash \textbf{if}\,\texttt{(new C().m())}\ \textbf{true else false} \Rightarrow \mathit{true}$$
$$\emptyset \Vdash \textbf{if}\,\texttt{(new C().m())}\ \textbf{true else false} \Rightarrow \mathit{false}$$

and no other values can be returned.

Case 2 (a), (b) and (c): There exists a unique value $\mathbb{v}$ s.t. $\emptyset \Vdash e \Rightarrow \mathbb{v}$

We consider three possible cases (a), (b), and (c), where the returned value is finite (a), or infinite but regular (b), or infinite and non regular (c). For case (a), if C is the class of case 1, then the expression **if (new C().m()) true else true** trivially evaluates to the unique value *true* (although with two different valid proof trees). For case (b), let us consider a program with the following declarations (where M, L, and n are abbreviations for Main, List, and next, respectively):

```
class M extends Object{L m(){new L(this.m())}}
class L extends Object{L n;}
```

The main expression **new** M().m() evaluates to a unique value which is an infinite but regular object of class L; Figure 6 shows the unique valid proof tree for $\emptyset \Vdash$ **new** M().m() $\Rightarrow obj(\texttt{L}, [\texttt{n} \mapsto \mathbb{v}])$; such a tree is infinite, but regular. The proof tree is valid if and only if the following proposition holds:

$$\Pi \Vdash \textbf{new}\ \texttt{L(this.m())} \Rightarrow \mathbb{v} \text{ iff } \Pi \Vdash \textbf{new}\ \texttt{L(this.m())} \Rightarrow obj(\texttt{L}, [\texttt{n} \mapsto \mathbb{v}])$$

with $\Pi = \texttt{this} \mapsto obj(\texttt{M}, [\,])$. Such a proposition cannot be satisfied by finite values, but holds for the unique infinite regular value $\mathbb{v}$ s.t. $\mathbb{v} = obj(\texttt{L}, [\texttt{n} \mapsto \mathbb{v}])$.

In the conventional nominal type system the return type τ of method m in M must verify $\texttt{L} \leq \tau$, since the body of the method returns a new instance of class L, but also $\tau \leq \texttt{L}$, since the formal parameter of the implicit constructor of L has the same type as field n; therefore, similarly to what happens in the CBS, there exists only one possible return type: L. This example shows that if rules are interpreted coinductively, but values can only be finite, then the soundness claim proved in Section 5, (that is, any well-typed expression evaluates to a value) does not hold.

Finally, for case (c), let us consider the following class declarations:

$$\dfrac{\dfrac{}{\emptyset \Vdash \texttt{new M()} \Rightarrow obj(\texttt{M}, [\,])} \qquad \dfrac{\dfrac{}{\Pi \Vdash \texttt{this} \Rightarrow obj(\texttt{M}, [\,])} \qquad \dfrac{\vdots}{\Pi \Vdash \texttt{new L(this.m())} \Rightarrow \mathbb{v}}}{\Pi \Vdash \texttt{this.m()} \Rightarrow \mathbb{v}}}{\emptyset \Vdash \texttt{new M().m()} \Rightarrow obj(\texttt{L}, [\texttt{n} \mapsto \mathbb{v}])}$$

$\Pi \Vdash \texttt{new L(this.m())} \Rightarrow obj(\texttt{L}, [\texttt{n} \mapsto \mathbb{v}])$

Fig. 6. Proof of $\emptyset \Vdash \texttt{new M().m()} \Rightarrow obj(\texttt{L}, [\texttt{n} \mapsto \mathbb{v}])$, with $\mathbb{v} = obj(\texttt{L}, [\texttt{n} \mapsto \mathbb{v}])$, $\Pi = \texttt{this} \mapsto obj(\texttt{M}, [\,])$

```
class Nat extends Object{ }
class Z extends Nat{ }
class NZ extends Nat{Nat p;}
class M extends Object{L m(Nat e){new L(e,this.m(new NZ(e)))}}
class L extends Object{Nat e; L n;}
```

Then, the expression **new** `M().m(new Z())` is well-typed and evaluates to the unique infinite and non regular value $\mathbb{v}_0$ where

$$\mathbb{v}_i = obj(\texttt{L}, [\texttt{e} \mapsto \mathbb{u}_i, \texttt{n} \mapsto \mathbb{v}_{i+1}]) \text{ for all } i \in \mathbb{N}$$
$$\mathbb{u}_0 = obj(\texttt{Z}, [\,]) \qquad \mathbb{u}_i = obj(\texttt{NZ}, [\texttt{p} \mapsto \mathbb{u}_{i-1}]) \text{ for all } i \in \mathbb{N} \setminus \{0\}$$

Figure 7 shows the non regular valid proof tree for the judgment, defined in terms of a countably infinite set of equations whose solutions are valid proof trees $\Rrightarrow\!\nabla_i$, for all $i \in \mathbb{N}$. This example shows that if rules are interpreted coinductively, but values can only be regular, then the soundness claim proved in Section 5, (that is, any well-typed expression evaluates to a value) does not hold.

$$\Rrightarrow\!\nabla_i = \dfrac{\dfrac{}{\Pi_i \Vdash \texttt{e} \Rightarrow \mathbb{u}_i} \qquad \dfrac{\dfrac{}{\Pi_i \Vdash \texttt{this} \Rightarrow obj(\texttt{M}, [\,])} \qquad \dfrac{\dfrac{}{\Pi_i \Vdash \texttt{e} \Rightarrow \mathbb{u}_i}}{\Pi_i \Vdash \texttt{new NZ(e)} \Rightarrow \mathbb{u}_{i+1}} \qquad \Rrightarrow\!\nabla_{i+1}}{\Pi_i \Vdash \texttt{this.m(new NZ(e))} \Rightarrow \mathbb{v}_{i+1}}}{\Pi_i \Vdash \texttt{new L(e,this.m(new NZ(e)))} \Rightarrow \mathbb{v}_i}$$

$$\dfrac{\dfrac{}{\emptyset \Vdash \texttt{new M()} \Rightarrow obj(\texttt{M}, [\,])} \qquad \dfrac{}{\emptyset \Vdash \texttt{new Z()} \Rightarrow \mathbb{u}_0} \qquad \Rrightarrow\!\nabla_0}{\emptyset \Vdash \texttt{new M().m(new Z())} \Rightarrow \mathbb{v}_0}$$

Fig. 7. Proof of $\emptyset \Vdash \texttt{new M().m(new Z())} \Rightarrow \mathbb{v}_0$, with $\Pi_i = \texttt{this} \mapsto obj(\texttt{M}, [\,]), \texttt{e} \mapsto \mathbb{u}_i$ for all $i \in \mathbb{N}$

Case 3: There exist no values $\mathtt{v}$ s.t. $\emptyset \Vdash e \Rightarrow \mathtt{v}$

If `C` is as in case 1 (that is, **new** `C().m()` does not terminate), then the expression **if**`(new C().m())` **true**`.m()` **else true**`.m()` does not evaluate to any value; this is a direct consequence of the fact that no rules are applicable for the expression **true**`.m()` since **true** does not evaluate to an object value. The main difference with the previous two cases is that here the expression to be evaluated cannot be typed in any type system insensitive to non-termination. Indeed, in the conventional nominal type system defined in Section 5 all examples except for this are well-typed. This example shows the main difference between the ISS and the CBS: in the former, there exist ill-typed expressions whose evaluation does not terminate (that is, does not get stuck), whereas in the latter all ill-typed expressions do not evaluate to a value.

Soundness of CBS w.r.t. ISS. We prove now that the CBS is sound w.r.t.the ISS. More precisely, if $\emptyset \Vdash e \Rightarrow \mathtt{v}$, then in the ISS either e diverges (that is, e does not reduce to a normal form), or e reduces in zero or more steps to a value v s.t. $\emptyset \Vdash v \Rightarrow \mathtt{v}$. In other words, we are guaranteed that the evaluation of an expression will never get stuck in the ISS whenever it returns a value in the CBS. Under this point of view the CBS plays a role similar to that of a type system; indeed, to prove this property we use the standard proof technique based on the progress and subject reduction properties. Such a property tells us an important fact: type soundness of a type system can be equivalently proved in terms of the CBS, instead of the ISS. If soundness holds in terms of the CBS, then it holds in terms of the ISS as well, by virtue of the soundness property of the CBS w.r.t. the ISS we are going to prove.

The progress and subject reduction properties can be proved routinely (see the companion technical report), the former by induction on e, the latter by induction on the rules defining ISS. Proof by coinduction is only needed for the substitution lemma.

Theorem 3 (Progress). *If $\emptyset \Vdash e \Rightarrow \mathtt{v}$, then either e is a value, or there exists e' s.t. $e \to e'$.*

Subject reduction relies on the following restricted form of substitution lemma which suffices for proving Theorem 4.

Lemma 1 (Substitution). *If $\overline{x}^n \mapsto \overline{\mathtt{v}}^n \Vdash e \Rightarrow \mathtt{v}$, and for all $i = 1..n$ $\emptyset \Vdash v_i \Rightarrow \mathtt{v}_i$, then $\emptyset \Vdash e[\overline{x}^n \mapsto \overline{v}^n] \Rightarrow \mathtt{v}$.*

Theorem 4 (Subject reduction). *If $\emptyset \Vdash e \Rightarrow \mathtt{v}$, and $e \to e'$, then $\emptyset \Vdash e' \Rightarrow \mathtt{v}$.*

Corollary 1. *If $\emptyset \Vdash e \Rightarrow \mathtt{v}$, $e \stackrel{*}{\to} e'$, and e' is a normal form, then e' is a value, and $\emptyset \Vdash e' \Rightarrow \mathtt{v}$.*

Proof. By induction on the number n of steps needed to reduce e to e'. If $n = 0$, then $e = e'$, and trivially $\emptyset \Vdash e' \Rightarrow \mathtt{v}$; furthermore, since e' is a normal form, by progress (Theorem 3) e' is a value. If $n > 0$, then there exists e'' s.t. $e \to e''$, and e'' reduces to e' in $n - 1$ steps. By subject reduction (Theorem 4) $\emptyset \Vdash e'' \Rightarrow \mathtt{v}$, then we conclude by inductive hypothesis.

5 Type Systems

To make the proof of soundness simpler and more modular, we first define a standard inductive nominal type system for our reference language, and then we derive from it a coinductive nominal type system, and prove that if an expression is well-typed in the inductive type system, than it is assigned the same type in the coinductive one. In other words, the inductive type system is sound w.r.t. the coinductive one; we conjecture that in fact the two systems are equivalent (hence, the coinductive system is sound w.r.t. the inductive one as well), but here we prove only the only implication we are interested in. In this way, soundness of the inductive type system in terms of the CBS can be directly derived from soundness of the coinductive type system in terms of the CBS (prove in Section 6).

Auxiliary Definitions. Besides functions *fields* and *meth*, already used for defining both the ISS and the CBS, the typing rules are based on the following auxiliary functions/operators, whose definition is straightforward (see the companion technical report). The standard subtyping relation $\leq$ between nominal types; the *override* predicate s.t. $override(c, m, \overline{\tau}^n, \tau)$ holds iff $meth(c', m)$ is undefined or $meth(c', m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau'$, $\overline{\tau'}^n \leq \overline{\tau}^n$, and $\tau \leq \tau'$, with c' direct superclass of c; the join operator $\vee$ which computes the least upper bound $\vee(\tau_1, \tau_2)$ of two types τ_1 and τ_2 (this is always defined since inheritance is single, and *bool* is a subtype of the top type `Object`).

Typing Rules. The typing rules, which can be found in Figure 8, are quite standard. A type environment Γ is a finite sequence $\overline{x_i}^n{:}\overline{\tau}^n$, where all variables $\overline{x_i}^n$ are distinct, denoting a finite function mapping variables to types ($\emptyset$ denotes the empty type environment, $dom(\Gamma)$ the domain of Γ). Rules (*pro*), (*cla*), and (*met*) define well-typed programs, classes, and methods, respectively. The other rules define well-typed expressions w.r.t. a given type environment. Let us recall that, similarly to what happens for the operational semantics, all typing judgments are implicitly indexed over a class table containing all needed information on the classes declared in the program.

Membership Relation. To prove soundness of the type system w.r.t. the CBS, we first define a relation $\mathbb{v} \in \tau$ between the CBS values and nominal types: intuitively, such a relation defines the intended semantics of types as set of values [6]. Such a relation is coinductively defined by the following rules:

$$\text{(TOP)}\ \frac{}{\mathbb{v} \in \texttt{Object}} \qquad \text{(BOOL)}\ \frac{\mathbb{v} = \mathit{false} \text{ or } \mathbb{v} = \mathit{true}}{\mathbb{v} \in \mathit{bool}}$$

$$\text{(OBJ)}\ \frac{\forall\, i = 1..n.\mathbb{v}_i \in \tau_i \quad c \leq c' \quad \mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n}{\mathit{obj}(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n]) \in c'}$$

The membership relation is easily extended to environments and type environments:

$$\Pi \in \Gamma \Leftrightarrow dom(\Gamma) \subseteq dom(\Pi) \text{ and } \forall\, x \in dom(\Gamma).\Pi(x) \in \Gamma(x).$$

$$(pro)\ \frac{\forall i = 1..n. \vdash cd_i{:}\diamond \quad \emptyset \vdash e{:}\tau}{\vdash \overline{cd}^n\ e{:}\diamond} \qquad (cla)\ \frac{\forall i = 1..k. c \vdash md_i{:}\diamond \quad \mathit{fields}(c)\ \text{defined}}{\vdash \mathbf{class}\ c\ \mathbf{extends}\ c'\ \{\ \overline{fd}^n\ \overline{md}^k\ \}{:}\diamond}$$

$$(met)\ \frac{\mathtt{this}{:}c, \overline{x}^n{:}\overline{\tau}^n \vdash e{:}\tau \quad \tau \leq \tau_0 \quad \mathit{override}(c, m, \overline{\tau}^n, \tau_0)}{c \vdash \tau_0\ m(\overline{\tau}^n\ \overline{x}^n)\ \{e\}{:}\diamond}$$

$$(var)\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x{:}\tau} \qquad (fal)\ \frac{}{\Gamma \vdash \mathbf{false}{:}\mathit{bool}} \qquad (tru)\ \frac{}{\Gamma \vdash \mathbf{true}{:}\mathit{bool}}$$

$$(new)\ \frac{\forall i = 1..n. \Gamma \vdash e_i{:}\tau_i \quad \mathit{fields}(c) = \overline{\tau'}^n\ \overline{f}^n \quad \forall i = 1..n. \tau_i \leq \tau'_i}{\Gamma \vdash \mathbf{new}\ c(\overline{e}^n){:}c}$$

$$(fld)\ \frac{\Gamma \vdash e{:}c \quad \mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n \quad 1 \leq i \leq n}{\Gamma \vdash e.f_i{:}\tau_i} \qquad (if)\ \frac{\Gamma \vdash e{:}\mathit{bool} \quad \Gamma \vdash e_1{:}\tau_1 \quad \Gamma \vdash e_2{:}\tau_2}{\Gamma \vdash \mathbf{if}\ (e)\ e_1\ \mathbf{else}\ e_2{:}\vee(\tau_1, \tau_2)}$$

$$(inv)\ \frac{\forall i = 0..n. \Gamma \vdash e_i{:}\tau_i \quad \mathit{meth}(\tau_0, m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau \quad \forall i = 1..n. \tau_i \leq \tau'_i}{\Gamma \vdash e_0.m(\overline{e}^n){:}\tau}$$

Fig. 8. Nominal type system

Coinductive Type System. The coinductive type system is derived from the inductive one defined in Figure 8 as follows:

- all rules for typing expressions are interpreted coinductively (rules for well-typed programs, classes, and methods can be indifferently interpreted inductively or coinductively, since they are not recursive);
- all rules are unchanged, except for rule (inv) which is modified in (co-inv):

$$(co\text{-}inv)\ \frac{\begin{array}{l}\forall i = 0..n. \Gamma \Vdash e_i{:}\tau_i \quad \mathtt{this}{:}\tau_0, \overline{x}^n{:}\overline{\tau}^n \Vdash e{:}\tau' \\ \mathit{meth}(\tau_0, m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau \quad \forall i = 1..n. \tau_i \leq \tau'_i, \tau' \leq \tau\end{array}}{\Gamma \Vdash e_0.m(\overline{e}^n){:}\tau}$$

Rule (*co-inv*) is clearly not compositional: instead of type checking a method once for all, and using subtyping and type safe overriding (as happens in the inductive system), the coinductive type system checks a method body, not only when it is declared (rule (*cla*)), but also whenever it is called. However, from a more theoretical point of view, the coinductive type system is a step closer to the CBS. Of course the type system must be interpreted coinductively, otherwise typechecking of recursive methods would always fail. Consider for instance case 2 (b) presented in Section 4. The judgment $\emptyset \Vdash$ **new** `M().m()`:`L` can be derived only with an infinite proof tree, as depicted in Figure 9. Note that the proof tree is isomorphic to the proof tree for $\emptyset \Vdash$ **new** `M().m()` $\Rightarrow \mathit{obj}(\mathtt{L}, [\mathtt{n} \mapsto \mathtt{v}])$ shown in Figure 6.

We can now prove soundness of the inductive type system w.r.t. the coinductive one.

$$\dfrac{\dfrac{\vdots}{\texttt{this:M}\Vdash\texttt{this:M}} \quad \dfrac{\vdots}{\texttt{this:M}\Vdash\textbf{new L(this.m())}\texttt{:L}}}{\dfrac{\texttt{this:M}\Vdash\texttt{this.m():L}}{\dfrac{}{\emptyset\Vdash\textbf{new M()}\texttt{:M}} \quad \dfrac{\texttt{this:M}\Vdash\textbf{new L(this.m())}\texttt{:L}}{\emptyset\Vdash\textbf{new M().m()}\texttt{:L}}}}$$

Fig. 9. Proof for $\emptyset\Vdash$**new** `M().m():L`

The following lemmas are instrumental to the proof of the theorem 5 that follows; in the claims of all lemmas we implicitly assume that judgments refer to a well-typed program. All omitted proofs can be found in the companion technical report.

Lemma 2. *If* $\tau_1' \leq \tau_1$ *and* $\tau_2' \leq \tau_2$ *then* $\vee(\tau_1', \tau_2') \leq \vee(\tau_1, \tau_2)$.

Lemma 3. *If* $\mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n$, *and* $c' \leq c$, *then* $\mathit{fields}(c') = \overline{\tau}^m\ \overline{f}^m$ *with* $n \leq m$.

Lemma 4. *If* $\mathit{meth}(c, m) = \overline{\tau}^n\ \overline{x}^n.e{:}\tau$, *then there exist* c', τ' *s.t.* $c \leq c'$, $\tau' \leq \tau$ *and* $\texttt{this}{:}c', \overline{x}^n{:}\overline{\tau}^n \vdash e{:}\tau'$.

Lemma 5. *If* $\overline{x}^n{:}\overline{\tau}^n \vdash e{:}\tau$ *and for all* $i = 1..n$ $\tau_i' \leq \tau_i$, *then there exists* τ' *s.t.* $\tau' \leq \tau$, *and* $\overline{x}^n{:}\overline{\tau'}^n \vdash e{:}\tau'$.

Lemma 6. *The subtyping relation is transitive: if* $\tau_1 \leq \tau_2$ *and* $\tau_2 \leq \tau_3$, *then* $\tau_1 \leq \tau_3$.

Theorem 5. *Let* $\overline{cd}^n$ *be well-typed class declarations. If* $\Gamma \vdash e{:}\tau$ *in* $\overline{cd}^n$, *then* $\Gamma\Vdash e{:}\tau$ *in* $\overline{cd}^n$.

Proof. By coinduction on the rules defining the judgment $\_\Vdash\_{:}\_$, and case analysis on e. The only interesting case is when e is a method invocation $e_0.m(\overline{e}^n)$, since for all other cases the rules of the two type systems coincide. If $\Gamma \vdash e{:}\tau$, then by definition of rule (*inv*) we have $\forall\, i = 0..n. \Gamma \vdash e_i{:}\tau_i$, $\mathit{meth}(\tau_0, m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau$, and $\forall\, i = 1..n. \tau_i \leq \tau_i'$. By lemma 4 there exists τ_0', τ' s.t. $\tau_0 \leq \tau_0'$, $\tau' \leq \tau$, and $\texttt{this}{:}\tau_0', \overline{x}^n{:}\overline{\tau'}^n \vdash e{:}\tau'$; therefore, by lemma 5 there exists τ'' s.t. $\tau'' \leq \tau'$, and hence by transitivity (lemma 6) $\tau'' \leq \tau$, and $\texttt{this}{:}\tau_0, \overline{x}^n{:}\overline{\tau}^n \vdash e{:}\tau''$. We conclude by coinductive hypothesis and by definition of rule (*co-inv*).

6 Proof of Soundness

In this section we prove the middle implication shown in Figure 1, which is the core of our result: soundness of the coinductive type system in terms of the CBS. Finally, by virtue of the soundness of the inductive type system w.r.t. the coinductive one (proved in Section 5), and of the soundness of the CBS w.r.t.

the ISS (proved in Section 4), we can state soundness of the inductive type system in terms of the ISS as a simple corollary.

The following lemmas are instrumental to the proof of the theorem 6 that follows; in the claims of all lemmas we implicitly assume that judgments refer to a well-typed program. All omitted proofs can be found in the companion technical report.

Lemma 7. *$\tau_1 \leq \vee(\tau_1, \tau_2)$ and $\tau_2 \leq \vee(\tau_1, \tau_2)$.*

Lemma 8. *If $fields(c) = \overline{\tau}^n\ \overline{f}^n$, and $c \leq c'$, then $fields(c') = \overline{\tau}^m\ \overline{f}^m$ with $m \leq n$.*

Lemma 9 (Soundness of subtyping). *If $\mathbb{v} \in \tau$ and $\tau \leq \tau'$, then $\mathbb{v} \in \tau'$.*

Lemma 10. *If $meth(c, m) = \overline{\tau}^n\ \overline{x}^n.e{:}\tau$ and $c' \leq c$, then $meth(c', m) = \overline{\tau'}^n\ \overline{x'}^n.e'{:}\tau'$ where for all $i = 1..n\ \tau_i \leq \tau'_i$ and $\tau' \leq \tau$.*

To prove that the coinductive type system is sound w.r.t. the CBS, we coinductively define a *concretization relation* $\mathcal{R}_\gamma$ between valid proof trees ${}_{\Vdash}\nabla \in VPT_:$ for $\Gamma \Vdash e{:}\tau$ and (possibly non valid) proof trees ${}_{\Rightarrow}\nabla \in PT_\Rightarrow$ for $\Pi \Vdash e \Rightarrow \mathbb{v}$, and show that for any valid proof tree ${}_{\Vdash}\nabla$ for $\Gamma \Vdash e{:}\tau$ and any $\Pi \in \Gamma$, there exists a value $\mathbb{v}$ and a valid proof tree ${}_{\Rightarrow}\nabla$ for $\Pi \Vdash e \Rightarrow \mathbb{v}$ s.t. ${}_{\Vdash}\nabla\ \mathcal{R}_\gamma\ {}_{\Rightarrow}\nabla$, and $\mathbb{v} \in \tau$.

Definition 9. *A relation $\mathcal{R} \subseteq VPT_: \times PT_\Rightarrow$ is a concretization iff the following constraints are satisfied:*

- $\dfrac{}{\Gamma \Vdash x{:}\tau}\ \mathcal{R}\ \dfrac{}{\Pi \Vdash x \Rightarrow \mathbb{v}}$ *iff $\Pi \in \Gamma$, and* $ok(\textsc{var})\left(\dfrac{}{\Pi \Vdash x \Rightarrow \mathbb{v}}\right)$
- $\dfrac{}{\Gamma \Vdash \mathbf{false}{:}bool}\ \mathcal{R}\ \dfrac{}{\Pi \Vdash \mathbf{false} \Rightarrow false}$ *iff $\Pi \in \Gamma$*
- $\dfrac{}{\Gamma \Vdash \mathbf{true}{:}bool}\ \mathcal{R}\ \dfrac{}{\Pi \Vdash \mathbf{true} \Rightarrow true}$ *iff $\Pi \in \Gamma$*
- $\dfrac{\overline{{}_{\Vdash}\nabla}^n}{\Gamma \Vdash \mathbf{new}\ c(\overline{e}^n){:}c}\ \mathcal{R}\ \dfrac{\overline{{}_{\Rightarrow}\nabla}^n}{\Pi \Vdash \mathbf{new}\ c(\overline{e}^n) \Rightarrow \mathbb{v}}$ *iff for all $i = 1..n\ {}_{\Vdash}\nabla_i\ \mathcal{R}\ {}_{\Rightarrow}\nabla_i$, $\Pi \in \Gamma$, and* $ok(\textsc{new})\left(\dfrac{\overline{{}_{\Rightarrow}\nabla}^n}{\Pi \Vdash \mathbf{new}\ c(\overline{e}^n) \Rightarrow \mathbb{v}}\right)$
- $\dfrac{{}_{\Vdash}\nabla}{\Gamma \Vdash e.f{:}\tau}\ \mathcal{R}\ \dfrac{{}_{\Rightarrow}\nabla}{\Pi \Vdash e.f \Rightarrow \mathbb{v}}$ *iff ${}_{\Vdash}\nabla\ \mathcal{R}\ {}_{\Rightarrow}\nabla$, $\mathbb{v} \in \tau$, $\Pi \in \Gamma$, and* $ok(\textsc{fld})\left(\dfrac{{}_{\Rightarrow}\nabla}{\Pi \Vdash e.f \Rightarrow \mathbb{v}}\right)$
- $\dfrac{{}_{\Vdash}\nabla\ {}_{\Vdash}\nabla_1\ {}_{\Vdash}\nabla_2}{\Gamma \Vdash \mathbf{if}\ (e)\ e_1\ \mathbf{else}\ e_2{:}\tau}\ \mathcal{R}\ \dfrac{{}_{\Rightarrow}\nabla\ {}_{\Rightarrow}\nabla_1}{\Pi \Vdash \mathbf{if}\ (e)\ e_1\ \mathbf{else}\ e_2 \Rightarrow \mathbb{v}}$ *iff ${}_{\Vdash}\nabla\ \mathcal{R}\ {}_{\Rightarrow}\nabla$, ${}_{\Vdash}\nabla_1\ \mathcal{R}\ {}_{\Rightarrow}\nabla_1$, $root({}_{\Rightarrow}\nabla) = \Pi \Vdash e \Rightarrow true$, $\mathbb{v} \in \tau$, $\Pi \in \Gamma$, and* $ok(\textsc{ift})\left(\dfrac{{}_{\Rightarrow}\nabla\ {}_{\Rightarrow}\nabla_1}{\Pi \Vdash \mathbf{if}\ (e)\ e_1\ \mathbf{else}\ e_2 \Rightarrow \mathbb{v}}\right)$

– $\dfrac{{}_{\Vdash}\nabla\ {}_{\Vdash}\nabla_1\ {}_{\Vdash}\nabla_2}{\Gamma\Vdash\textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2{:}\tau}\ \mathcal{R}\ \dfrac{{}_{\Rightarrow}\nabla\ {}_{\Rightarrow}\nabla_2}{\Pi\Vdash\textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2\Rightarrow\mathtt{v}}$ *iff* ${}_{\Vdash}\nabla\ \mathcal{R}\ {}_{\Rightarrow}\nabla$, ${}_{\Vdash}\nabla_2\ \mathcal{R}\ {}_{\Rightarrow}\nabla_2$, $root({}_{\Rightarrow}\nabla) = \Pi\Vdash e\Rightarrow false$, $\mathtt{v}\in\tau$, $\Pi\in\Gamma$, *and* $ok_{(\text{IFF})}\left(\dfrac{{}_{\Rightarrow}\nabla\ {}_{\Rightarrow}\nabla_2}{\Pi\Vdash\textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2\Rightarrow\mathtt{v}}\right)$

– $\dfrac{{}_{\Vdash}\nabla_0\ {}_{\Vdash}\overline{\nabla}^n\ {}_{\Vdash}\nabla}{\Gamma\Vdash e_0.m(\overline{e}^n){:}\tau}\ \mathcal{R}\ \dfrac{{}_{\Rightarrow}\nabla_0\ {}_{\Rightarrow}\overline{\nabla}^n\ {}_{\Rightarrow}\nabla}{\Pi\Vdash e_0.m(\overline{e}^n)\Rightarrow\mathtt{v}}$ *iff for all* $i = 0..n$ ${}_{\Vdash}\nabla_i\mathcal{R}_{\Rightarrow}\nabla_i$, ${}_{\Vdash}\nabla\mathcal{R}_{\Rightarrow}\nabla$, $\mathtt{v}\in\tau$, $\Pi\in\Gamma$, *and* $ok_{(\text{INV})}\left(\dfrac{{}_{\Rightarrow}\nabla_0\ {}_{\Rightarrow}\overline{\nabla}^n\ {}_{\Rightarrow}\nabla}{\Pi\Vdash e_0.m(\overline{e}^n)\Rightarrow\mathtt{v}}\right)$.

The function $\mathcal{F}$ corresponding to the recursive definition of concretization relation is trivially monotone on the complete lattice defined by the power set of $VPT_{:}\times PT_{\Rightarrow}$, therefore by the Tarski-Knaster theorem there exists the greatest concretization relation, denoted by $\mathcal{R}_\gamma$. However, the Tarski-Knaster theorem does not provide any guarantee that for any valid proof tree ${}_{\Vdash}\nabla$ for $\Gamma\Vdash e{:}\tau$ and any $\Pi\in\Gamma$, there exists a value $\mathtt{v}$ and a valid proof tree ${}_{\Rightarrow}\nabla$ for $\Pi\Vdash e\Rightarrow\mathtt{v}$ s.t. ${}_{\Vdash}\nabla\ \mathcal{R}_\gamma\ {}_{\Rightarrow}\nabla$, and $\mathtt{v}\in\tau$. To prove such a property we need to apply the Kleene theorem; indeed, $\mathcal{F}$ also preserves infima of descending chains in the same complete lattice, hence, the concretization relation $\mathcal{R}_\gamma$ is defined by $\inf\{\mathcal{F}^n(\top)\mid n\in\mathbb{N}\}$, where $\top$ denotes the top element of the lattice defined by the power set of $VPT_{:}\times PT_{\Rightarrow}$, that is, the relation associating any valid proof tree for the judgment $\_\Vdash\_{:}\_$, with any proof tree for the judgment $\_\Vdash\_\Rightarrow\_$. We abbreviate $\mathcal{F}^n(\top)$ with $\mathcal{R}^n_\gamma$, hence $\mathcal{R}^0_\gamma=\top$.

As an example, we have that ${}_{\Vdash}\nabla^e\mathcal{R}_\gamma{}_{\Rightarrow}\nabla^e$, where ${}_{\Vdash}\nabla^e$ and ${}_{\Rightarrow}\nabla^e$ are the proof trees defined in Figure 9 and 6, respectively. The easiest way to prove this fact is to show that there exists a concretization relation $\mathcal{R}$ s.t. ${}_{\Vdash}\nabla^e\ \mathcal{R}\ {}_{\Rightarrow}\nabla^e$; this can be achieved by considering the finite relation that associates each subtree of ${}_{\Vdash}\nabla^e$ (including ${}_{\Vdash}\nabla^e$ itself), with the corresponding subtree[3] of ${}_{\Rightarrow}\nabla^e$ (including ${}_{\Rightarrow}\nabla^e$ itself); it is immediate to verify that such a relation is a concretization.

However, when proving soundness the proof tree for the CBS is unknown, and therefore its existence is proved by showing that it can be obtained as the limit of a Cauchy sequence in a complete metric space. Therefore, to better understand the proof that will follow, it is instructive to show how the proof tree ${}_{\Rightarrow}\nabla^e$ can actually be built from ${}_{\Vdash}\nabla^e$ by using the Kleene construction. We assume that the expression is evaluated in a program where the only available classes are `M` and `L` as declared for case 2 (b) in Section 4, and we use ellipses ... as a wildcard.

– ${}_{\Vdash}\nabla^e\mathcal{R}^0_\gamma{}_{\Rightarrow}\nabla$ for any ${}_{\Rightarrow}\nabla\in PT_{\Rightarrow}$.
– ${}_{\Vdash}\nabla^e\mathcal{R}^1_\gamma{}_{\Rightarrow}\nabla$ for any ${}_{\Rightarrow}\nabla\in PT_{\Rightarrow}$ s.t. ${}_{\Rightarrow}\nabla=$

$$\dfrac{\dfrac{\cdots}{\Pi\Vdash\textbf{new}\ \texttt{M()}\Rightarrow\mathtt{v}}\quad\dfrac{\cdots}{\texttt{this}\mapsto\mathtt{v}\Vdash\textbf{new}\ \texttt{L(this.m())}\Rightarrow\mathtt{v}_0}}{\Pi\Vdash\textbf{new}\ \texttt{M().m()}\Rightarrow\mathtt{v}_0}$$

[3] We recall that the two trees are isomorphic; furthermore, they have a finite number of subtrees, since they are regular.

where $\Pi \in \emptyset$ (hence, Π can be any value environment), $\mathtt{v}$ can be any value, and $\mathtt{v}_0 \in \mathtt{L}$, hence for all $i \in \mathbb{N}$, $\mathtt{v}_i = obj(\mathtt{L}, [\mathtt{n} \mapsto \mathtt{v}_{i+1}])$, therefore $\mathtt{v}_0$ is the unique value s.t. $\mathtt{v}_0 = obj(\mathtt{L}, [\mathtt{n} \mapsto \mathtt{v}_0])$ and $\mathtt{v}_i = \mathtt{v}_{i+1}$, for all $i \in \mathbb{N}$. Note that, since we have assumed that `M` and `L` are the only available classes of the program, there exists only one possible subtype of `L`, namely `L` itself, and the equations above can be directly derived by applying membership rule OBJ. Therefore, for this particular case we get the returned value (in this particular case it is unique) just at the first iteration; however, for getting the corresponding valid proof tree all iterations have to be considered.
We proceed with the next iteration, to show how at each step the obtained proof trees are better approximations of a valid proof tree.

– $\Vdash^e \mathcal{R}^2_\gamma \Rightarrow$ for any $\Rightarrow \in PT_\Rightarrow$ s.t. $\Rightarrow =$

$$\dfrac{\dfrac{}{\Pi \Vdash \mathbf{new}\ \mathtt{M()} \Rightarrow obj(\mathtt{M}, [\,])} \quad \dfrac{\dfrac{\dfrac{\dots}{\mathtt{this} \mapsto obj(\mathtt{M}, [\,]) \Vdash \mathtt{this.m()} \Rightarrow \mathtt{v}_0}}{\mathtt{this} \mapsto obj(\mathtt{M}, [\,]) \Vdash \mathbf{new}\ \mathtt{L(this.m())} \Rightarrow \mathtt{v}_0}}{\Pi \Vdash \mathbf{new}\ \mathtt{M().m()} \Rightarrow \mathtt{v}_0}$$

where $\Pi \in \emptyset$ (hence, Π can be any value environment). Note that, by virtue of the equation $\mathtt{v}_0 = obj(\mathtt{L}, [\mathtt{n} \mapsto \mathtt{v}_0])$, the evaluation of **new** `L(this.m())` and of `this.m()` returns the same vale $\mathtt{v}_0$.

It can be easily proved by standard induction over n that $\Vdash^e \mathcal{R}^n_\gamma \Rightarrow^e$, for all $n \in \mathbb{N}$, where $\Rightarrow^e$ is the valid proof tree defined in Figure 6; since $\mathcal{R}_\gamma$ is the greatest lower bound of $\{\mathcal{R}^n_\gamma | n \in \mathbb{N}\}$, we obtain that $\Vdash^e \mathcal{R}_\gamma \Rightarrow^e$.

To prove the main claim from which soundness of the coinductive type system w.r.t. the CBS can be derived, we need to define a complete metric space of proof trees for the CBS.

We first define the metric of value environment. We recall that a value environment is a finite partial function mapping variables to values, and that values are finitely branching trees with infinite paths (hence, they form a complete metric space with the distance d_T of Definition 1).

Proposition 2. *The set of value environments forms a complete metric space when equipped with the distance d_Π defined as follows:*

$$d_\Pi(\Pi_1, \Pi_2) = \begin{cases} 1 & \text{if } dom(\Pi_1) \neq dom(\Pi_2) \\ \max(\{0, d_T(\Pi_1(x), \Pi_2(x)) \mid x \in D\}) & \text{if } D = dom(\Pi_1) = dom(\Pi_2) \end{cases}$$

Proof. See the companion technical report.

Proposition 3. *The set of pairs of value environments and values forms a complete metric space when equipped with the distance $d_{\Pi,\mathtt{v}}$ defined as follows:*

$$d_{\Pi,\mathtt{v}}((\Pi_1, \mathtt{v}_1), (\Pi_2, \mathtt{v}_2)) = \max\{d_\Pi(\Pi_1, \Pi_2), d_T(\mathtt{v}_1, \mathtt{v}_2)\}$$

Proof. A well known property of product metric spaces that can be easily checked. From Proposition 2 one can easily deduce that $0 \leq d_{\Pi,\mathtt{v}}((\Pi_1, \mathtt{v}_1), (\Pi_2, \mathtt{v}_2)) \leq 1$, since $d_{\Pi,\mathtt{v}}((\Pi_1, \mathtt{v}_1), (\Pi_2, \mathtt{v}_2)) \in \{0\} \cup \{2^{-c} \mid c \in \mathbb{N}\}$.

Let j be the judgment $\Pi \Vdash e \Rightarrow \mathtt{v}$, then $ev(j)$ and $exp(j)$ denote $(\Pi, \mathtt{v})$ and e, respectively; furthermore, $exp(\underset{\Rightarrow}{\nabla})$ denotes the tree t over expressions s.t. $dom(t) = dom(\underset{\Rightarrow}{\nabla})$, and for all $p \in dom(t)$ $t(p) = exp(\underset{\Rightarrow}{\nabla}(p))$.

Proposition 4. *The set $PT_{\Rightarrow}$ of proof trees for $\Pi \Vdash e \Rightarrow \mathtt{v}$ forms a complete metric space when equipped with the distance d_{∇} defined as follows:*

$$d_{\nabla}(\underset{\Rightarrow}{\nabla}_1, \underset{\Rightarrow}{\nabla}_2) = \max(\{2^{-c}\} \cup S) \text{ where}$$

$S = \{2^{-k} \cdot d_{\Pi,\mathtt{v}}(ev(\underset{\Rightarrow}{\nabla}_1(p)), ev(\underset{\Rightarrow}{\nabla}_2(p))) \mid p \in \mathbb{N}^k \cap dom(\underset{\Rightarrow}{\nabla}_1), 0 \leq k < c\}$
$c = shtp(exp(\underset{\Rightarrow}{\nabla}_1), exp(\underset{\Rightarrow}{\nabla}_2))$, *that is,* $c = \min\{n \in \mathbb{N} \mid p \in \mathbb{N}^n, exp(\underset{\Rightarrow}{\nabla}_1(p)) \neq_{\perp} exp(\underset{\Rightarrow}{\nabla}_2(p))\}$ *(see Proposition 1 for the definition of shtp and the related notation).*

Proof. See the companion technical report.

Let us consider the Kleene approximations $\mathcal{R}^i_{\gamma}$ $(i \in \mathbb{N})$ of the concretization relation $\mathcal{R}_{\gamma}$. Then the following lemma holds, where we assume that judgments are indexed over a class table corresponding to a sequence of well-typed classes $\overline{cd}^n$.

Lemma 11 (Substitution). *Let $\underset{\Vdash}{\nabla}$ be a valid proof tree for $\Gamma \Vdash e{:}\tau$, and $\underset{\Rightarrow}{\nabla}$ a (not necessarily valid) proof tree for $\Pi \Vdash e \Rightarrow \mathtt{v}$. For all $n \in \mathbb{N}$, if the following facts hold:*

1. $\underset{\Vdash}{\nabla} \mathcal{R}^n_{\gamma} \underset{\Rightarrow}{\nabla}$
2. $\Pi, \Pi' \in \Gamma$, $d_{\Pi}(\Pi, \Pi') \leq 2^{-n}$
3. *there exists* $\underset{\Rightarrow}{\nabla}'$ *s.t.* $\underset{\Vdash}{\nabla} \mathcal{R}^{n+1}_{\gamma} \underset{\Rightarrow}{\nabla}'$ *and* $d_{\nabla}(\underset{\Rightarrow}{\nabla}, \underset{\Rightarrow}{\nabla}') \leq 2^{-n}$

then there exists a proof tree $\underset{\Rightarrow}{\nabla}''$ for $\Pi' \Vdash e \Rightarrow \mathtt{v}'$ s.t. $\underset{\Vdash}{\nabla} \mathcal{R}^{n+1}_{\gamma} \underset{\Rightarrow}{\nabla}''$ and $d_{\nabla}(\underset{\Rightarrow}{\nabla}, \underset{\Rightarrow}{\nabla}'') \leq 2^{-n}$.

Proof. The proof is by induction on n, and by case analysis on the expression e.

Lemma 12. *For all $n \in \mathbb{N}$, $\underset{\Vdash}{\nabla} \in VPT_{:}$, and $\underset{\Rightarrow}{\nabla} \in PT_{\Rightarrow}$, if $\underset{\Vdash}{\nabla} \mathcal{R}^n_{\gamma} \underset{\Rightarrow}{\nabla}$, then there exists $\underset{\Rightarrow}{\nabla}'$ s.t. $\underset{\Vdash}{\nabla} \mathcal{R}^{n+1}_{\gamma} \underset{\Rightarrow}{\nabla}'$, and $d_{\nabla}(\underset{\Rightarrow}{\nabla}, \underset{\Rightarrow}{\nabla}') \leq 2^{-n}$.*

Proof. The proof is by induction on n, and by case analysis on the expression e.

Basis: If $n = 0$, then by definition $\mathcal{R}^0_{\gamma} = \top$, and, hence, $\underset{\Vdash}{\nabla} \mathcal{R}^0_{\gamma} \underset{\Rightarrow}{\nabla}$ for all $\underset{\Vdash}{\nabla} \in VPT_{:}$, and $\underset{\Rightarrow}{\nabla} \in PT_{\Rightarrow}$; therefore we have to show that there exists $\underset{\Rightarrow}{\nabla}'$ s.t. $\underset{\Vdash}{\nabla} \mathcal{R}^1_{\gamma} \underset{\Rightarrow}{\nabla}'$. Let us consider the case where $e = e_0.m(\overline{e}^n)$ (for all other cases the proof is analogous). If $\underset{\Vdash}{\nabla}$ is a proof tree for $\Gamma \Vdash e_0.m(\overline{e}^n){:}\tau$, then by rule (*co-inv*) we have that $\Gamma \Vdash e_0{:}\tau_0$ and $meth(\tau_0, m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau$. By definition of membership, there always exist Π and $\mathtt{v}$ s.t. $\Pi \in \Gamma$, and $\mathtt{v} \in \tau$, hence we can pick any Π and $\mathtt{v}$ s.t. $\Pi \in \Gamma$, and $\mathtt{v} \in \tau$, and build the following (not necessarily valid) proof tree:

$$\underset{\Rightarrow}{\nabla}' = \dfrac{\forall\, i = 0..n. \dfrac{}{\Pi \Vdash e_i \Rightarrow \mathtt{v}_i} \quad \dfrac{}{\mathtt{this} \mapsto \mathtt{v}_0, \overline{x}^n \mapsto \overline{\mathtt{v}}^n \Vdash e \Rightarrow \mathtt{v}}}{\Pi \Vdash e_0.m(\overline{e}^n) \Rightarrow \mathtt{v}}$$

with $\mathbb{v}_0 = obj(\tau_0, [\ldots])$ and $meth(\tau_0, m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau$. Clearly, $\Vdash\nabla\mathcal{R}^1_\gamma\Rightarrow\nabla'$, since by definition 9, and by definition of $\mathcal{R}^1_\gamma$,

$$\frac{\Vdash\nabla_0\ \overline{\Vdash\nabla}^n\ \Vdash\nabla}{\Gamma\Vdash e_0.m(\overline{e}^n){:}\tau}\ \mathcal{R}^1_\gamma\ \frac{\Rightarrow\nabla_0\ \overline{\Rightarrow\nabla}^n\ \Rightarrow\nabla}{\Pi\Vdash e_0.m(\overline{e}^n)\Rightarrow\mathbb{v}}$$

iff for all $i = 0..n$ $\Vdash\nabla_i\mathcal{R}^0_\gamma\Rightarrow\nabla_i$, $\Vdash\nabla\mathcal{R}^0_\gamma\Rightarrow\nabla$, $\mathbb{v}\in\tau$, $\Pi\in\Gamma$, and $ok(\text{INV})\left(\dfrac{\Rightarrow\nabla_0\ \overline{\Rightarrow\nabla}^n\ \Rightarrow\nabla}{\Pi\Vdash e_0.m(\overline{e}^n)\Rightarrow\mathbb{v}}\right)$.

Finally, by Proposition 4 we have that $d_\nabla(\Rightarrow\nabla, \Rightarrow\nabla') \leq 2^{-0} = 1$ for all $\Rightarrow\nabla, \Rightarrow\nabla' \in PT_\Rightarrow$.

Inductive Step: we have to prove that for all $n \geq 1$, $\Vdash\nabla\mathcal{R}^{n-1}_\gamma\Rightarrow\nabla \Rightarrow \exists\Rightarrow\nabla'$ s.t. $\Vdash\nabla\mathcal{R}^n_\gamma\Rightarrow\nabla'$, and $d_\nabla(\Rightarrow\nabla, \Rightarrow\nabla') \leq 2^{-n+1}$ implies $\Vdash\nabla\mathcal{R}^n_\gamma\Rightarrow\nabla \Rightarrow \exists\Rightarrow\nabla'$ s.t. $\Vdash\nabla\mathcal{R}^{n+1}_\gamma\Rightarrow\nabla'$, and $d_\nabla(\Rightarrow\nabla, \Rightarrow\nabla') \leq 2^{-n}$.

As for the basis, we consider the case where $e = e_0.m(\overline{e}^n)$ (for all other cases the proof is analogous). Therefore let us assume that $\Vdash\nabla\mathcal{R}^n_\gamma\Rightarrow\nabla$, where $\Vdash\nabla$ is a valid proof tree for $\Gamma\Vdash e_0.m(\overline{e}^n){:}\tau$. By rule (*co-inv*) we have

$$\Vdash\nabla = \frac{\Vdash\nabla_0\ \overline{\Vdash\nabla}^n\ \Vdash\nabla'}{\Gamma\Vdash e_0.m(\overline{e}^n){:}\tau}$$

with $meth(\tau_0, m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau$, $\forall i = 1..n.\tau_i \leq \tau_i'$, $\tau' \leq \tau$, and where $\forall i = 1..n.root(\Vdash\nabla_i) = \dfrac{\vdots}{\Gamma\Vdash e_i{:}\tau_i}$, $root(\Vdash\nabla') = \dfrac{\vdots}{\texttt{this}{:}\tau_0, \overline{x}^n{:}\overline{\tau}^n\Vdash e{:}\tau'}$.

Since $\Vdash\nabla\mathcal{R}^n_\gamma\Rightarrow\nabla$, by Definition 9 and by definition of $\mathcal{R}^n_\gamma$ we have

$$\Rightarrow\nabla = \frac{\Rightarrow\nabla_0\ \overline{\Rightarrow\nabla}^n\ \Rightarrow\nabla'}{\Pi\Vdash e_0.m(\overline{e}^n)\Rightarrow\mathbb{v}}$$

and for all $i = 0..n$ $\Vdash\nabla_i\mathcal{R}^{n-1}_\gamma\Rightarrow\nabla_i$, $\Vdash\nabla'\mathcal{R}^{n-1}_\gamma\Rightarrow\nabla'$, $\mathbb{v}\in\tau$, $\Pi\in\Gamma$, and $ok(\text{INV})(\Rightarrow\nabla)$. Then by inductive hypothesis we have that there exist $\Rightarrow\nabla'_0, \ldots, \Rightarrow\nabla'_n$ and $\Rightarrow\nabla''$ s.t. for all $i = 0..n$ $\Vdash\nabla_i\mathcal{R}^n_\gamma\Rightarrow\nabla'_i$, $d_\nabla(\Rightarrow\nabla_i, \Rightarrow\nabla'_i) \leq 2^{-n+1}$, $\Vdash\nabla'\mathcal{R}^n_\gamma\Rightarrow\nabla''$, $d_\nabla(\Rightarrow\nabla', \Rightarrow\nabla'') \leq 2^{-n+1}$. Therefore we have that for all $i = 0..n$ $root(\Rightarrow\nabla'_i) = \Pi_i\Vdash e_i \Rightarrow \mathbb{v}_i$, $root(\Rightarrow\nabla'') = \Pi'\Vdash e \Rightarrow \mathbb{v}'$, with $\Pi_i \in \Gamma$, $\Pi' \in (\texttt{this}{:}\tau_0, \overline{x}^n{:}\overline{\tau}^n)$, $\mathbb{v}_i \in \tau_i$ (hence, $\mathbb{v}_0 = obj(\tau_0, [\ldots])$) and $\mathbb{v}' \in \tau$. By lemma 11 we can derive from $\Rightarrow\nabla'_i$ $(i = 0..n)$ and from $\Rightarrow\nabla''$ the proof trees $\Rightarrow\nabla''_i$ $(i = 0..n)$ and $\Rightarrow\nabla'''$ s.t. for all $i = 0..n$ $\Vdash\nabla_i\mathcal{R}^n_\gamma\Rightarrow\nabla''_i$, $d_\nabla(\Rightarrow\nabla_i, \Rightarrow\nabla''_i) \leq 2^{-n+1}$, $\Vdash\nabla'\mathcal{R}^n_\gamma\Rightarrow\nabla'''$, $d_\nabla(\Rightarrow\nabla', \Rightarrow\nabla''') \leq 2^{-n+1}$, and $root(\Rightarrow\nabla''_i) = \Pi\Vdash e_i \Rightarrow \mathbb{v}'_i$, $root(\Rightarrow\nabla''') = \texttt{this} \mapsto \mathbb{v}'_0, \overline{x}^n \mapsto \overline{\mathbb{v}'}^n\Vdash e \Rightarrow \mathbb{v}''$

Finally, the proof tree

$$\Vdash\bar{\nabla} = \frac{\Vdash\nabla''_0\ \overline{\Vdash\nabla''}^n\ \Vdash\nabla'''}{\Gamma\Vdash e_0.m(\overline{e}^n){:}\tau}$$

is s.t. $\Vdash\nabla\mathcal{R}^{n+1}_\gamma\Rightarrow\bar{\nabla}$, and $d_\nabla(\Rightarrow\nabla, \Rightarrow\bar{\nabla}) \leq 2^{-n}$, by definition of $\mathcal{R}^{n+1}_\gamma$ and d_∇.

We can now state the main result.

Theorem 6. *Let* $\overline{cd}^n$ *be well-typed class declarations. If* $\Gamma \Vdash e{:}\tau$ *and* $\Pi \in \Gamma$ *in* $\overline{cd}^n$*, then there exists* $\mathtt{v}$ *s.t.* $\Pi \Vdash e \Rightarrow \mathtt{v}$ *and* $\mathtt{v} \in \tau$ *in* $\overline{cd}^n$.

Proof. Let ${}_{\Vdash}\nabla$ be a proof tree for $\Gamma \Vdash e{:}\tau$; directly from lemma 12 we deduce that it is possible to build a Cauchy sequence $({}_{\Rightarrow}\nabla_i)_{i\in\mathbb{N}}$ of proof trees s.t. ${}_{\Vdash}\nabla \mathcal{R}^i_\gamma {}_{\Rightarrow}\nabla_i$ for all $i \in \mathbb{N}$; by Proposition 4, such a sequence has a certain limit ${}_{\Rightarrow}\nabla$, s.t. ${}_{\Vdash}\nabla \mathcal{R}_\gamma {}_{\Rightarrow}\nabla$, which is a valid proof tree for $\Pi \Vdash e \Rightarrow \mathtt{v}$, with $\mathtt{v} \in \tau$. Note that, if the metric space of proof trees is not complete, then we could not deduce that the sequence $({}_{\Rightarrow}\nabla_i)_{i\in\mathbb{N}}$ has a limit; indeed, if we restrict the CBS to finite or regular values, then it is not possible to define a complete metric space, and, therefore, the sequence $({}_{\Rightarrow}\nabla_i)_{i\in\mathbb{N}}$ has no limit, and the claim of soundness does not hold, as already observed in the examples 2 (b) and (c) in Section 4.

Soundness of the inductive type system in terms of the CBS and of the ISS can be derived as two simple corollaries.

Corollary 2. *Let* $\overline{cd}^n$ *be well-typed class declarations. If* $\Gamma \vdash e{:}\tau$*, and* $\Pi \in \Gamma$ *in* $\overline{cd}^n$*, then there exists* $\mathtt{v}$ *s.t.* $\Pi \Vdash e \Rightarrow \mathtt{v}$ *and* $\mathtt{v} \in \tau$ *in* $\overline{cd}^n$.

Proof. The theorem is a straightforward corollary of Theorems 5 and 6.

Corollary 3. *If* $\emptyset \vdash e{:}\tau$*,* $e \xrightarrow{*} e'$*, and* e' *is a normal form, then* e' *is a value.*

Proof. Direct from corollaries 2 and 1.

Such a corollary is sufficient for guaranteeing the soundness of the type system in terms of the ISS: a well-typed expression can never get stuck in the ISS. However, by adding the following property (that can be proved easily), we can also deduce that the value e' is s.t. $\emptyset \vdash e'{:}\tau'$ with $\tau' \leq \tau$.

Proposition 5. *If* $\emptyset \Vdash v \Rightarrow \mathtt{v}$*, and* $\mathtt{v} \in \tau$*, then* $\emptyset \vdash v{:}\tau'$*, with* $\tau' \leq \tau$.

We can now prove the generalization of Corollary 3.

Corollary 4. *If* $\emptyset \vdash e{:}\tau$*,* $e \xrightarrow{*} e'$*, and* e' *is a normal form, then* e' *is a value and* $\emptyset \vdash e'{:}\tau'$ *with* $\tau' \leq \tau$.

Proof. By Corollary 2 we know also that $\mathtt{v} \in \tau$, and by Corollary 1 we know that $\emptyset \Vdash e' \Rightarrow \mathtt{v}$, hence we can conclude by Proposition 5.

We end this section by providing a generic scheme to be adopted for proving soundness of a type system in terms of the CBS of a language. We consider the case where one would like also to relate the CBS to the ISS, and derive from such a relation a standard soundness claim expressed in terms of the ISS.

We assume that the ISS is defined by a reduction relation $e_1 \rightarrow e_2$, and a set of values v (which are a subset of expressions in normal form), and the CBS is defined by a judgment $\Pi \Vdash e \Rightarrow \mathtt{v}$, where Π is an environment associating variables with values, and $\mathtt{v}$ is a value (all definitions are expected to be coinductive). The type system is defined by a judgment $\Gamma \vdash e{:}\tau$, where Γ is a type environment associating variables with types, and τ is a type, and subtyping

$\tau_1 \leq \tau_2$ and membership $\mathtt{v} \in \tau$ (which is easily extended to environments) are defined. Finally, a coinductive type system, defined by a judgment $\Gamma \Vdash e{:}\tau$, can be routinely defined from the inductive one.

Then the following properties have to be proved:

1. If $\emptyset \Vdash e \Rightarrow \mathtt{v}$, then either e is a value, or there exists e' s.t. $e \to e'$.
2. If $\emptyset \Vdash e \Rightarrow \mathtt{v}$, and $e \to e'$, then $\emptyset \Vdash e' \Rightarrow \mathtt{v}$.
3. If $\Gamma \vdash e{:}\tau$, then $\Gamma \Vdash e{:}\tau$.
4. If $\Gamma \Vdash e{:}\tau$, and $\Pi \in \Gamma$, then there exists $\mathtt{v}$ s.t. $\Pi \Vdash e \Rightarrow \mathtt{v}$ and $\mathtt{v} \in \tau$.
5. If $\emptyset \Vdash v \Rightarrow \mathtt{v}$, and $\mathtt{v} \in \tau$, then $\emptyset \vdash v{:}\tau'$, with $\tau' \leq \tau$.

We stress again that primitive properties 1 and 2 involve ISS and CBS only and, hence, can be proved once per each language, and reused for any type system.

From the claim above one can derive the following properties:

- If $\emptyset \vdash e{:}\tau$, $e \xrightarrow{*} e'$, and e' is a normal form, then e' is a value. Derivable from claims 1,2, 3, and 4.
- If $\emptyset \vdash e{:}\tau$, $e \xrightarrow{*} e'$, and e' is a normal form, then e' is a value and $\emptyset \vdash e{:}\tau'$ with $\tau' \leq \tau$. Derivable if claim 5 holds as well.

7 Conclusion and Related Work

We have shown that it is possible to prove soundness of a conventional inductive and nominal type system for a Java-like language in terms of a coinductive big-step operational semantics obtained by interpreting coinductively the rules of the standard big-step semantics. We have also suggested a generic scheme, where parts of the proofs can be reused, to be adopted for proving soundness of a type system in terms of the CBS of a language. The key point of the result is that infinite (including non regular) values have to be considered, otherwise the claim fails. Infinite values allow the definition of a complete metric space of proof trees for the CBS, which ensures that every well-typed expression evaluates into a value in the CBS, even in case of non-termination.

We have also shown that the CBS can be regarded as the concretization of a coinductive type system that can be directly derived from the standard inductive type system. Beside making the proof of soundness clearer, this fact also reveals how coinduction is related to the inductive type system.

With respect to the traditional one, the proposed approach may seem overly more complex, although big-step operational semantics tend to be simpler than small-step ones, especially when one wants to model more significant subsets of a real language. The main source of complexity comes from the proofs in Section 6, and from the fact that coinduction is less intuitive than induction. It would be worth investigating whether coalgebraic techniques could be used, to avoid using complete metric spaces. However, we hope that the provided definitions and proofs can be easily adapted for other type systems and languages.

The pioneering work of Milner and Tofte [14] is one of the first where coinduction is used for proving consistency of the type system and the big-step semantics

of a simple functional language; however rules are interpreted inductively, and the semantics does not capture diverging evaluations.

In their work Leroy and Grall [13] analyze two kinds of coinductive big-step operational semantics for the call-by-value λ-calculus, study their relationships with the small-step and denotational semantics, and their suitability for compiler correctness proofs. Besides the fact that here we consider a Java-like language, the main contribution of this paper w.r.t. Leroy and Grall's work is showing that by interpreting coinductively a standard big-step operational semantics, soundness of a standard nominal type system can be proved. We could prove such a result because (1) in our semantics not only evaluation rules are interpreted coinductively, but also the definition of values, and (2) the absence of first-class functions in our language makes the treatment simpler. Leroy and Grall show that a similar soundness claim does not hold in their setting; we conjecture that the only reason for that consists in the fact that in their coinductive semantics values are defined inductively (hence are finite), rather than coinductively (that is, infinite). It would be interesting to investigate whether soundness holds for the λ-calculus when values are defined coinductively.

Kusmierek and Bono propose a different approach and prove type soundness w.r.t. an inductive big-step operational semantics; their proposal is centered on the idea of tracing the intermediate steps of a program execution with a partial derivation-search algorithm which deterministically computes the value and the proof tree of evaluation judgments. Similar approaches, although their corresponding semantics are not deterministic, are those of Ager [1] and Stoughton [18].

Nakata and Uustalu [16,15] define a coinductive trace-based semantics, whose main aim, however, is formal verification of non-terminating programs.

Finally we would like to mention the work by Ernst et al. [10] where a soundness result w.r.t. a big-step operational semantics is proved thanks to a coverage lemma ensuring that errors do not prevent expressions from evaluating to a result. Such a result is achieved by introducing a finite evaluation relation indexed over natural numbers. A terminating expression is one for which there exists a natural number n such that the finite evaluation indexed by n returns a value (which may include also the usual runtime errors). However, in our approach type soundness can be proved without introducing extra rules for dealing with runtime errors generation and propagation, and finite evaluations.

References

1. Ager, M.S.: From Natural Semantics to Abstract Machines. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 245–261. Springer, Heidelberg (2005)
2. Amadio, R., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems 15(4), 575–631 (1993)
3. Ancona, D.: Coinductive big-step operational semantics for type soundness of Java-like languages. In: Formal Techniques for Java-like Programs (FTfJP 2011), pp. 5:1–5:6. ACM (2011)

4. Ancona, D., Corradi, A., Lagorio, G., Damiani, F.: Abstract Compilation of Object-Oriented Languages into Coinductive CLP(X): Can Type Inference Meet Verification? In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 31–45. Springer, Heidelberg (2011)
5. Ancona, D., Lagorio, G.: Coinductive Type Systems for Object-Oriented Languages. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 2–26. Springer, Heidelberg (2009)
6. Ancona, D., Lagorio, G.: Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In: Montanari, A., Napoli, M., Parente, M. (eds.) Proceedings of GandALF 2010. Electronic Proceedings in Theoretical Computer Science, vol. 25, pp. 214–223 (2010)
7. Ancona, D., Lagorio, G.: Idealized coinductive type systems for imperative object-oriented programs. RAIRO - Theoretical Informatics and Applications 45(1), 3–33 (2011)
8. Arnold, A., Nivat, M.: The metric space of infinite trees. Algebraic and topological properties. Fundamenta Informaticae 3, 445–476 (1980)
9. Courcelle, B.: Fundamental properties of infinite trees. Theoretical Computer Science 25, 95–169 (1983)
10. Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: POPL, pp. 270–282 (2006)
11. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (2001)
12. Kusmierek, J.D.M., Bono, V.: Big-step operational semantics revisited. Fundam. Inform. 103(1-4), 137–172 (2010)
13. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation 207, 284–304 (2009)
14. Tofte, M., Milner, R.: Co-induction in relational semantics. Theoretical Computer Science 87(1), 209–220 (1990)
15. Nakata, K., Uustalu, T.: Trace-Based Coinductive Operational Semantics for While. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 375–390. Springer, Heidelberg (2009)
16. Nakata, K., Uustalu, T.: A Hoare Logic for the Coinductive Trace-Based Big-Step Semantics of While. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 488–506. Springer, Heidelberg (2010)
17. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive Logic Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 330–345. Springer, Heidelberg (2006)
18. Stoughton, A.: An operational semantics framework supporting the incremental construction of derivation trees. Electr. Notes Theor. Comput. Sci. 10 (1997)

Static Sessional Dataflow

Dominic Duggan and Jianhua Yao

Department of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07030, USA
{dduggan,jyao1}@stevens.edu

Abstract. Sessional dataflow provides a compositional semantics for dataflow computations that can be scheduled at compile-time. The interesting issues arise in enforcing static flow requirements in the composition of actors, ensuring that input and output rates of actors on related channels match, and that cycles in the composition of actors do not introduce deadlock. The former is ensured by flowstates, a form of behavior type that constrains the firing behavior of dataflow actors. The latter is ensured by causalities, a form of constraints that record dependencies in the firing behavior. This article considers an example variant of the sessional dataflow approach for dataflow applications, expressing known ideas from signal processing in a compositional fashion.

1 Introduction

Dataflow has an honored tradition in declarative parallel programming [12,10]. It has renewed significance today, given the importance attached to deterministic parallelism as a way of coping with the challenges of scalable parallel programming. Many of the applications of parallel processing are in stream processing, e.g., streaming multimedia data, again motivating interest in dataflow processing. Part of the challenge of dataflow processing is in scheduling the execution of dataflow graphs without unbounded buffering of data between actors in the net. In signal processing, synchronous dataflow has enjoyed some success for multi-rate applications, with many variations of the basic idea developed over the years [13].

The purpose of sessional dataflow is to provide a compositional semantics for dataflow computations that can be scheduled at compile-time. To explain why compositionality is important, in synchronous dataflow and its variants, a dataflow graph is described in terms of atomic actors, and flow edges connecting them. A compositional semantics allows both atomic actors, and subnets resulting from the composition of actors, to be viewed uniformly as dataflow actors. Compositionality is obviously important for scaling dataflow programming. The interesting issues arise in enforcing static flow requirements in the composition of actors, ensuring that input and output rates of actors on related channels match, and that cycles in the composition of actors do not introduce deadlock. Ultimately the purpose of sessional dataflow is to support dynamic operations on subnets, including update and reconfiguration, while ensuring that assumptions underlying static scheduling are not violated by these operations.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 484–508, 2012.

In the embedded systems and digital signal processing community, a very useful class of restricted Kahn networks has been identified, the so-called *synchronous dataflow (SDF)* [13] networks. SDF networks assumed fixed static input and output rates for actors when they fire in a dataflow network. Such networks are important because they can be scheduled statically by the compiler, ensuring a fixed upper bound on the amount of buffer space needed. Numerous extensions of SDF have been defined over the years, all pushing the envelope of expressivity while remaining in the space of dataflow applications that can be scheduled statically.

More recently, new domain-specific languages such as Streamit [16] have been defined, based on the principles of SDF, but also providing support for compiling programmer code to run on modern parallel architectures. Streamit is a dataflow language intended for the efficient compilation of stream processing programs. Its design rationale is that of *structured dataflow*. Rather than allowing arbitrary dataflow graphs, Streamit imposes structure on the graph, in order to facilitate compiler analysis and optimization. This is enforced by only supporting certain forms of nodes in a dataflow graph.

In this work, we consider another form of dataflow language. We go back to Kahn's original idea, of a dataflow network consisting of a collection of software components that communicate asynchronously via buffered message-passing. Deterministic parallelism is provided by preventing contention for message channels, and by preventing components from polling message channels. As with Kahn's original proposal, our core language is a conventional imperative language. We impose a type system on this language that ensures the static behavior required for compile-time scheduling, as with SDF. This allows the incremental construction of dataflow graphs as composite actors, based on connecting input and output channels in two graphs (that may be the same graph). There are two components to the compositional description: a notion of *flowstate*, analogous to typestate in object-oriented languages, that captures the static message-passing behavior of a process, and a notion of *causalities*, that allows the liveness of a dataflow graph to be checked compositionally even while channels are encapsulating in composite graphs.

In Sect. 2 we introduce sessional dataflow with the interface and implementation specification for a simple (atomic) actor. This relates the constraints on actor behavior reflected in an actor interface, with the actual internal implementation of the actor that is encapsulated by this interface. These two are not traditionally related in work on synchronous dataflow and its derivative techniques, where actors are treated as "black boxes" and the actor code left unanalyzed. In Sect. 3 we consider a compositional approach to building actor implementations, based on binding communication channels between two existing actors. As an exercise in statically ensuring that the composition of actors is well-formed, we require that the result of composing actors into a dataflow net, effectively a composite actor, be statically schedulable. We provide a type system in Sect. 4 and an operational semantics in Sect. 5. Sect. 6 considers related work, while Sect. 7 provides our conclusions.

2 Actors

In this section, we provide an example of a form of static dataflow that has been found useful for parallel processing in signal processing and embedded systems. So-called synchronous dataflow (SDF), more aptly named *static dataflow*, assumes that on each actor "firing," a statically fixed number of inputs is consumed on each input channel and a statically fixed number of outputs is produced on each output channel. With this restriction on a fixed number inputs and outputs for each firing, and a further requirement that there be no cyclic data dependencies in the graph connecting the actros, the scheduling of a synchronous dataflow net can be performed by the compiler. All scheduling decisions, and the amount of buffer space required, are determined at compile-time.

An actor specification needs a few other aspects to be defined. Although firing is atomic in SDF, our semantics for firing is implemented in a C-like core language, that consumes and produces messages one at a time. For modeling the states of an actor, we use the notion of *flowstate*, that tracks the state of an actor during a firing cycle. In addition, we need a specification of the input and output channels of an actor, that will subsequently be coupled with channels for other actors to form a dataflow network. An example of a specification for an actor in our type system is provided by the following:

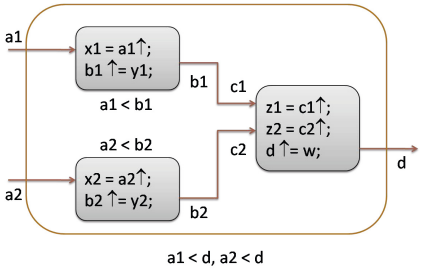
```
actor interface IActor
{
   in channel<float> a;
   in channel<float> b;
   out channel<float> c;
   causality a < c, b < c;
   flowstate 3'a, b, 2'c.
}
```

This is the expression of an actor type in our system. The type specifies input and output communication channels, and allowable communications on those channels using a flowstate specification. The flowstate rule in the example above requires that the actor consume three inputs on the `a` channel and one input on the `b` channel, and produces two outputs on the `c` channel. In what order should these inputs and outputs be performed? It is tempting to restrict firings so that all inputs are consumed before any outputs are produced, but once we compose actors into composite actors (dataflow nets), it is no longer possible to ensure this. Even if we restricted actors to only inputs or only outputs, but not a mixture of the two, we could still have scenarios where the consumption of an input in one actor depended on the production of an output in another actor. Therefore we must allow for aribtirary interleavings of inputs and outputs, while avoiding deadlock where for example an output channel and input channel are linked to the same underlying channel.

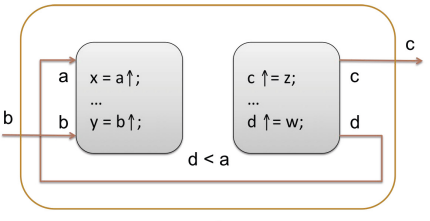
Therefore we enrich actor interfaces with a notion of *causalities*. This is demonstrated in the example above, where the causalities specify that outputs on channel `c` depend causally on inputs on channel `a` and on channel `b` (`a` $<$ `c` and `b` $<$ `c`). The exact number of inputs is provided by the multiplicities in the flowstate.

Why provide the causalities, since in this example all of the outputs depend on all of the inputs? This will be true in general for simple atomic actors, but may not be

true once we compose actors into nets of arbitrary complexity, with subnets running in parallel. The causalities are then useful to ensure that connecting an input and an output channel in such a composite actor does not introduce deadlock in the execution of the dataflow graph.



(a) Composite Actor with Internal Parallelism



(b) Composite Actor with Input depending on Output

Fig. 1. Sequential and Parallel Inputs

Fig. 1(a) and Fig. 1(b) demonstrate two composite actors. Fig. 1(a) provides a composite actor where two actors on the left consume inputs in parallel, and these are then consumed in a particular order by the actor on the right. We define causalities to reflect the fact that message sending is asynchronous, so in some sense the outputs of an actor, once their causally preceding input events occur, may occur in an indeterminate order. Fig. 1(b) provides another composite actor, one where the output on channel `c` must causally precede the input on channel `a`, since the output on internal channel `d` causally precedes the input on internal channel `a`. Outputs are parallel despite the fact that they are produced by a single sequential thread.

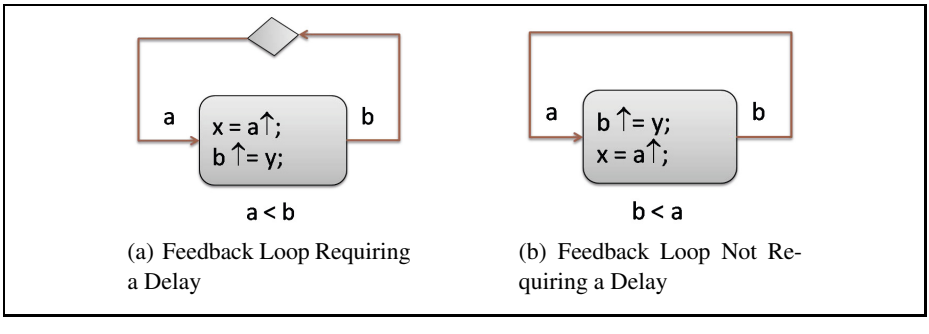
Our actor semantics is effectively a limited form of *cyclostatic dataflow* [2]. In the latter, an actor has a finite state control logic, and transitions between states of this logic on each firing. Its firing pattern then depends on the current state that it is in. Because we are providing specifications for input consumption and output production at the level of individual communication steps, the semantics of a "firing" in the traditional SDF sense is non-atomic, and we are essentially tracking a finite state control logic in the process of a firing. We consider how the language can be extended to cyclostatic firing at the end of Sect. 4.

The specifications of the input-output behavior make no reference to the actual values that are transmitted. For simplicity we have assumed that the channel types are fixed, so that only values of the declared type may be transmitted on a channel. In practice it may be beneficial to relax this restriction, though we defer these considerations to future work. We comment further on this and other future extensions in Sect. 6.

An implementation of this actor specification uses a conventional programming language to define the actor behavior, in the style of Kahn's original proposal for dataflow networks:

```
actor Actor implements IActor
{
   float x1, x2, x3, y1;
   loop {
      x1 = a↓; x2 = a↓; x3 = a↓; y1 = b↓;
      c ↑ (x1+x2); c ↑ (y1+x3);
   }
}
```

The definition of the actor implementation inherits the interface specification: input and output channels, causalities and flowstates. The operation for reading from an input channel `c` is denoted by $c\downarrow$, while the operation of writing a value (asynchronously) to an output channel is denoted by `c` $\uparrow$ v. The body of the actor is otherwise conventional C code, except for the top-level loop construct that guarantees the actor is always able to offer the specified firing behavior. The flowstate in the actor specification establishes behavior obligations for its execution, subject to the constraints imposed by the causalities.



(a) Feedback Loop Requiring a Delay

(b) Feedback Loop Not Requiring a Delay

Fig. 2. Causality and Feedback

Fig. 2 clarifies the point of the causalities. In general the issue is to detect when connecting two open channels in the same actor may introduce a cycle in the dependencies between the channels. To avoid this cycle which woud lead to deadlock, the connection of the channels is required to introduce a "delay," by filling the buffer for the channel with default initial values. Fig. 2(a) illustrates this, where the single output channel of an actor is connected to its input channel. The actor first reads from the input channel

a, before outputing to the output channel b. Note that we do not try to track data flow dependencies, our interest is in the control flow dependency from the consumption of input on a to the production of output on b. Suppose these two open channels are connected to the same shared channel, We assume an obvious causality from the output end of a shared channel to the input end, so this binding will introduce the causality $b < a$. This will introduce a cycle in the causalities, which we cannot allow. Therefore in this case the connection of two channels a and b on the same underlying channel must include a delay, as indicated by the diamond in Fig. 2(a).

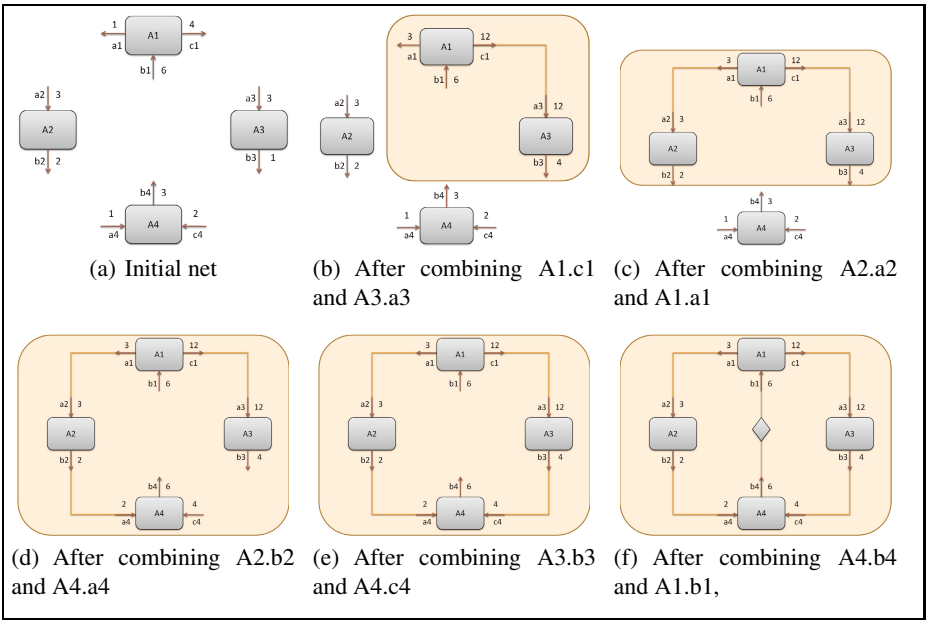
In the example in Fig. 2(b), on the other hand, the appending of data to the output buffer is done before input is performed. This results in the causality $b < a$ for the actor body. This does not necessarily mean that data flows from the output event to the input event, but there is at least a causal dependency, in that the occurrence of the output event is a prerequisite for the occurrence of the input event. When these input and output open channels are connected using the same shared input channel, then the output produced on the output channel does indeed propagate to the input channel to be consumed, but this is immaterial as far as scheduling is concerned, since communication is strictly internal to the actor. The causality $b < a$ that is added as a result of this binding of channels b and a adds no further constraints, since there is already a dependency from b to a, and no scheduling cycle is introduced, so a delay is not necessary.

3 Dataflow Nets

In the previous section, we considered the "programming-in-the-small" aspects of ensuring that an actor satisfied its behavior specification. In this section, we consider the "programming-in-the-large" aspect of ensuring that the composition of actors is in some sense well-formed. We consider the case of ensuring that the composition of synchronous dataflow actors is schedulable, based on the static firing rates of the actors.

In general, the approach to composition of actors is to provide a binary connection operation for linking the output data channel on one actor with the input data channel of another. We denote this operation by `connect(A.a,B.b)`. Here it is important to distinguish between open channels and shared channels. An *open channel* is one of the form described in the previous section, a channel that is declared in an actor interface, and referred to in an actor body by operations for consuming messages and appending messages to message buffers.

For deterministic semantics, it is important that there be no nondeterministic contention for access to a channel. For example, a nondeterministic merge might be provided by allowing multiple actors to send simultaneously to the same merge channel. Synchronization on access to the channel, performed by the compiler and runtime system, could ensure that the message append operations are atomic. However the order in which messages are appended would be nondeterministic, based on dynamic scheduling of actors and interleaving of their multithreaded executions. While nondeterministic merge is a useful operation in some cases, our intention is to establish a baseline that ensures deterministic execution, before considering later how to extend this with nondeterminism.



(a) Initial net (b) After combining A1.c1 and A3.a3 (c) After combining A2.a2 and A1.a1

(d) After combining A2.b2 and A4.a4 (e) After combining A3.b3 and A4.c4 (f) After combining A4.b4 and A1.b1,

Fig. 3. Dataflow nets

Our approach is to ensure exclusive access to a communication channel between two actors, the one actor sending on that channel and the other actor receiving on that channel. The connection operation `connect(A.a,B.b)` creates a new private communication channel, binds the `a` output channel on the `A` actor to the output part of this new private channel, and binds the `b` input channel on the `B` actor to the input part of this new private channel. We refer to such a private channel as a *shared channel*. Since (for now) we provide no way for an actor to send any of its communication channels to another actor, exclusive access by a pair of actors to a shared channel is ensured.[1]

What is the result of connecting actors? The semantics should be compositional, so that the connection of two actors should be indistinguishable to outside observers from a single actor. For synchronous dataflow, the only part of the outside interface of note for a combined actor is the remaining open channels after a connection, and the firing rates for those channels.

A type system for interconnection should guarantee that the actors being combined are in some sense compatible, so that the resulting actor is statically schedulable. The firing rates for actors that are interconnected may be different on the channel on which they are connected. The purpose of static scheduling is to match the relative input and output rates of communicating actors during execution. For connection of distinct

[1] Actor interfaces include polarity information about access to channels, and the connection operation requires that the accesses by the actors be complementary. It is indeed possible that the actors being connected are the same. The connection operation requires knowledge of when it is the case that the same actors are being connected, as we will see.

actors, it is always the case that their firing rates are compatible: Simply adjust their relative firing rates so that, on the connecting channel, their rates are the least common multiple of the original rates on that channel.

Fig. 3(a) depicts four characters: `A1`, `A2`, `A3` and `A4`. Each of these actors has open channels with firing rates. For example, actor `A1` has open channel `a1` with firing rate 1, open channel `b1` with firing rate 2, and open channel `c1` with firing rate 4. The names of these channels outside the actor are not significant, and we assume for simplicity that all channels are renamed apart.

In Fig. 3(b), the actors `A1` and `A3` are connected by binding the open channels `A1.c1` and `A3.a3` along an anonymous shared channel. Since the output rate of `A1` does not match the input firing rate of `A3`, we adjust the firing rates of the two actors to make them compatible. The connection of an output channel of `A1` to the input channel of `A3` causes the addition of the causality $\texttt{c1} < \texttt{a3}$. Although `a3` is elided in the resulting interface (since it has been bound to the output end of a communication channel), transitive closure of causalities adds the constraint $\texttt{b1} < \texttt{b3}$ (from $\texttt{b1} < \texttt{a3}$ and $\texttt{a3} < \texttt{b3}$).

Fig. 3(c) depicts the result of connecting the composite actor `connect(A1.c1,A3.a3)` with the actor `A2`, by binding the open channels `connect(A1.c1,A3.a3).a1` and `A2.a2` to a shared channel. In this case, the data rates of the actors on the respective open channels match, so no adjustment of firing rates is necessary.

Fig. 3(d) depicts the result of connecting the composite actor from Fig. 3(c) with the actor `A4`. This connection is done on the `b2` and `a4` open channels. Because of the difference in data rates, the firing rates for `A4` must be adjusted

At this point, we have only one (composite) actor, with some remaining open channels. We now complete the net by connecting different channels within the same actor. Fig. 3(e) depicts the result of connecting the `b3` and `c4` channels on this composite actor.

This actor can be completed by forming a feedback loop by connecting the output of the `b4` channel to the input of the `b1` channel. The types, and in particular the inclusion $\texttt{b4} < 24$, reveals the existing data dependency from the output channel `b4` to the input channel `b1`. This data dependency, revealed in the inclusion constraint in the interface, signals that linking the output channel to the input channel will in this case introduce a feedback loop. In order to ensure that the resulting net does not deadlock, a *delay* must be introduced in this new channel connection, as depicted by the diamond in Fig. 3(f).

What could possibly go wrong? Fig. 4 demonstrates how the checking during the composition of actors may fail. We have three actors, `B1`, `B2` and `B3`, with data rates as described in Fig. 4(a). We compose `B1` and `B2` by binding the channels `B1.a1` and `B2.a2` to a shared channel, adjusting the firing rates as necessary to make their data rates on the shared channel match. We repeat this exercise by composing the resultant composite actor with `B3`, binding the `b1` and `a3` channels to a shared channel. At this point, there is a problem: It is not possible to bind the remaining open channels `b2` and `b3` to each other, because they have different data rates.

To analyse the problem, we note that in general the scheduling of SDF actors can be thought of as the solution of a homogeneous system of linear equations. The independent variables in this system of equations are the number of times each actor fires on an iteration of the dataflow net, and the equations specify the constraint that the amount of

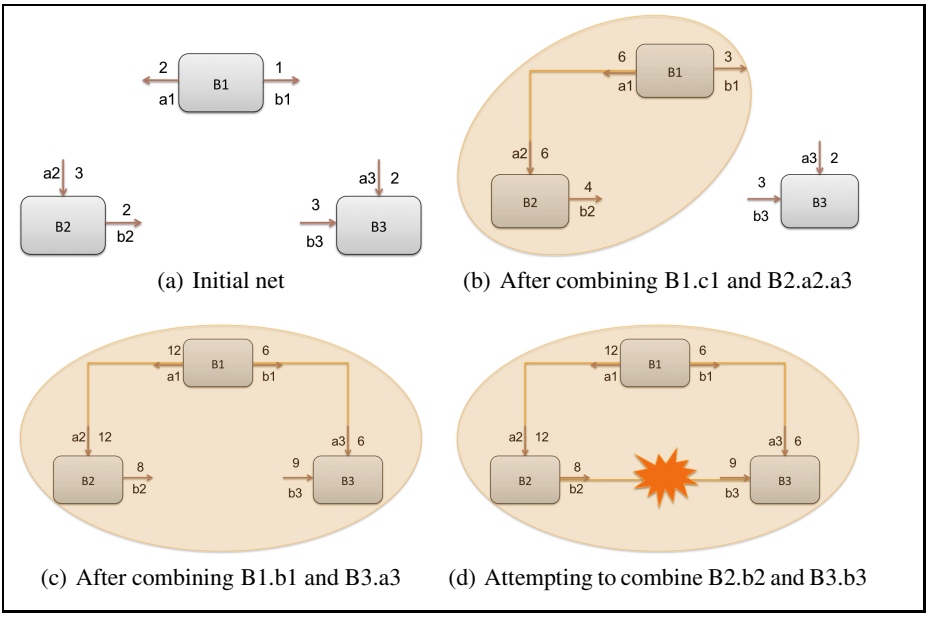


Fig. 4. Unschedulable dataflow net

outputs produced by each actor must match the number of inputs consumed, on each net iteration. If this constraint is not satisfied, then some message buffers will grow without bound during the execution of the net.

For the example above, we obtain the following system of equations, where $F_{\mathtt{B}i}$ denotes the number of firings of actor Bi on each iteration of the net:

$$\begin{aligned} 2 \cdot F_{\mathtt{B2}} - 3 \cdot F_{\mathtt{B1}} &= 0 \\ 2 \cdot F_{\mathtt{B3}} - 3 \cdot F_{\mathtt{B2}} &= 0 \\ F_{\mathtt{B3}} - 2 \cdot F_{\mathtt{B1}} &= 0 \end{aligned}$$

Solving for the independent variables by eliminating $F_{\mathtt{B3}}$, we obtain the equation:

$$3 \cdot F_{\mathtt{B2}} - 4 \cdot F_{\mathtt{B1}} = 0$$

Then using this and the first of the original equations to eliminate $F_{\mathtt{B2}}$, we obtain the equation:

$$9 \cdot F_{\mathtt{B1}} - 8 \cdot F_{\mathtt{B1}} = 0$$

The only solution to this equation is to fire B1, and therefore the other actors, zero times, i.e., to never run the dataflow net. This demonstrates the importance of being able to distinguish the cases when we are combining two distinct actors, and when we are connecting two open channels in the same actor. In the former case, when the actors are distinct, we can adjust the actors' firing rates so that the data rates on the connecting

channel are the least common multiple of the data rates on the corresponding open channels in the actors. In the latter case, when we are connecting channels on the same actor, the data rates must match on the corresponding open channels. If they do not, we do not have the extra degree of freedom that we have with distinct actors to adjust firing rates. Indeed, discovering different data rates on the channels is an important part of ensuring, at composition time, that we do not compose two actors into a composite actor (i.e., a dataflow net) that cannot be scheduled.

Variable aliasing is a potentially troublesome issue, for two reasons. First, communication on a channel changes the type of that channel, since the channel type reflects the communications that may be performed on that channel. we avoid the problem of variable aliasing by not allowing aliased references to communication channels. This is compatible with approaches such as for example session types that similarly constrain the bindings of variables to resources whose usage is tracked by linear or affine types.

A second potentially troublesome issue is with the connection of actors. As we have seen, connecting different actors provides a degree of freedom in adjusting the firing rates of the actors so that they match on the channels on which they are connrected. If we allow aliasing of actor references, then we must face the issue of how to deal with scenarios such as the following:

```
IActor2 f (IActor A, IActor B)
     return connect(A.a,B.b);
}
```

How can we prevent a scenario such as `f(A0,A0)`, for some actor `A0` that implements the `IActor` specification? This is a known issue in type systems for safe resource management, as discussed in Sect. 6. In our semantics, we avoid this issue because the `connect` operation makes copies of the two actors being composed. This is a potentially expensive operation, and a better choice of operations might split this into a connection operation that performed update in place on the argument actor specifications, and an explicit `clone` operation for explicitly making a copy of an actor. This choice however makes the actor connection operation a "strong update," modifying the interface of the original actor, which in turn requires ensuring that there be no references to the original actor remaining in the program (including aliases). Our copying semantics avoids this complication.

4 Type System

In this section we consider a core language to support the examples in the previous sections, including a type system to ensure valid program executions. We consider an operational semantics and type soundness in the next section. We name this kernel language $\mathbf{S_{SDF}}$. We only consider synchronous dataflow in this account, but we comment on the extension to the cyclostatic case at the end of this section.

The syntax of types is provided in Fig. 5. For simplicity we assume a single base type of `float`, for floating point values. Similarly we assume that only floating point values are exchanged between actors in each message exchange, so the channel type does not need to describe the type of data exchanged on the channel. Although the polyadic pi-calculus generalizes messages to include tuples of values, this is not necessary in our

$$
\begin{array}{rcl}
T \in \text{Type} & ::= & \mathsf{float} \mid AS \mid \mathsf{channel}\ \pi \\
AS \in \text{Actor sig} & ::= & \mathsf{actsig}(O, FS) \\
K \in \text{Causalities} & ::= & \{\} \mid \{a < b\} \mid K_1 \cup K_2 \\
O \in \text{Open channels} & ::= & \{\} \mid O_1 \cup O_2 \mid \{((\mathbf{c}, c) : \mathsf{channel}\ \pi)\} \\
\pi \in \text{Polarity} & ::= & \mathsf{+} \mid \mathsf{-} \mid \pm \\
ES \in \text{Event set} & ::= & \{\} \mid \{n \cdot a\} \mid ES_1 \uplus ES_2 \\
FS \in \text{Flowstate} & ::= & ES \mid \{FS \mid K\} \mid (FS_1; FS_2) \mid (FS_1 \parallel FS_2) \mid FS^* \mid FS^\omega
\end{array}
$$

Fig. 5. Abstract syntax of $\mathbf{S_{SDF}}$ Types

current framework because channels are private to a single sender and receiver. We do record polarity information for a channel, which records whether it can be used by that actor for input (polarity +), or for output (polarity −), or both (polarity ±).

The type of interest is that of actors. An *actor signature* has three parts, as we have seen:

1. A *causality set K* is a set of causality constraints between channels, of the form $a < b$, that reflects firing constraints between channels: If $a < b$, then in a firing of the actor or dataflow net, a communication on b depends on a communication on a. For simplicity, we assume that *all* communications on b depend on all communications on a. This set of dependencies must never contain a cycle.
2. A set of *open channels O*. Each element of this set is a triple $((\mathbf{c}, c) : T)$, recording for an "open" channel its channel type. This channel type has the form $\mathsf{channel}\ \pi$, where T is the type of data transmitted on the channel (we only allows floats to be transmitted in this article), and π is the polarity of the open channel. The channel has two names: its *internal name* c by which it is identified internally in the actor, and its *external name* $\mathbf{c}$ by which it is referenced when composing with other actors. We distinguish these names in order to allow renaming apart of internal channel names when actors are composed, without affecting the external interface. The internal and external names in different open channel bindings should obviously be distinct from each other. We define the domain of an open channel set as:
$$dom(O) = \{c \mid ((\mathbf{c}, c) : T) \in O\}.$$
We define the *external domain* of an open channel set as:
$$edom(O) = \{\mathbf{c} \mid ((\mathbf{c}, c) : T) \in O\}.$$
3. The *flowstate* of an actor records its expected firing behavior. The primitive form of an actor flowstate is an *event state*, a multiset of communication events $\{m_1 \cdot a_1, \ldots, m_k \cdot a_k\}$, where each a_i represents a communication event, either a sending of a message on a channel or a receipt of a message on a channel. In either case, we use the channel name to denote the event; whether it is an input or an output event can be determined by the channel's polarity. The multiplicities $m_1, \ldots, m_k$ record the number of occurrences of each event in an actor execution. The remaining forms of flowstate are used to describe the flowstate resulting from joining computations, either sequentially $(FS_1; FS_2)$ or in parallel $(FS_1 \parallel FS_2)$. Note that in the latter case

there may be communication dependencies between the actors running in parallel. The other forms of flowstates are for computations that can repeat an arbitrary number of times (FS^*) and that loop infinitely often (FS^ω). The latter corresponds to the top-level flowstate of an actor, primitive or composite.

$v \in$ Values	$::= n \mid a \mid x$	
$s \in$ Statement	$::= (\mathsf{var}\ x = e;\ s)$	Bind variables
	$\mid\ \mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2$	Conditional
	$\mid\ \mathsf{while}\ (v)\ s$	Loop
	$\mid\ \mathsf{loop}\ s$	Infinite loop
	$\mid\ \mathsf{fire}_K\ s$	Firing
	$\mid\ \mathsf{skip}$	Do nothing
	$\mid\ (s_1; s_2)$	Sequential
$e \in$ Expression	$::= f(v_1, \ldots, v_k)$	Builtin
	$\mid\ v_1 = v_2$	Assignment
	$\mid\ \mathsf{run}\ v$	Run a network
	$\mid\ c\downarrow$	Receive a message
	$\mid\ c\uparrow v$	Send a message
	$\mid\ \mathsf{actor}(O, s)$	Atomic actor
	$\mid\ \mathsf{connect}_{m,n}(v_1.\mathbf{c}_1, v_2.\mathbf{c}_2)$	Connect two actors
	$\mid\ \mathsf{connectSelf}_m(v_1.\mathbf{c}_1, v_1.\mathbf{c}_2)$	Connect within an actor
	$\mid\ \mathsf{connectSelfDelay}_m(v_1.\mathbf{c}_1, v_1.\mathbf{c}_2)$	Connect with delay

Fig. 6. Abstract syntax of $\mathbf{S_{SDF}}$ statements

Fig. 6 provides the abstract syntax for programs in $\mathbf{S_{SDF}}$. Values are numbers n, names a (for actors and channels, both allocated on the heap), and variables x. The syntax of statements includes a conditional[2], while loops, and a construct for doing nothing. We also have a construct (fire) for explicitly specifying the firing behavior of an actor body. The construct of interest is the binding statement, which introduces a new variable bound to the result of evaluating a definition. Some of the definitional expressions return a dummy value (the number 0). In these cases, the variable binding expression is used solely to sequence the computation. Why do we also include the sequencing of statements? Our expressional language is in A-normal form, but we have constructs such as the while loop and the conditional that do not fit the definition of an execution step in A-normal form. If we convert this language, both expressions and statements, into A-normal form, we obtain an exponential blow-up in code size unless we residualize the continuation of a conditional. We avoid these complications by including sequencing of statements.

The simplest form of definition is the invocation of builtin functions, presumably to perform arithmetic operations on numeric values. We also allow assignment, though only for values of base type, i.e., floating point values, because of the aliasing issue described in Sect. 3. As expected, we also have operations for receiving and sending messages.

[2] We are assuming that our language has no round-off error. Obviously a more complete language definition would include integers and Booleans.

$$\frac{}{\vdash_\Sigma \{\}\ \mathsf{ok}}\ \textsc{Env Empty} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad \Gamma \vdash_\Sigma T}{\vdash_\Sigma \Gamma, x:T\ \mathsf{ok}}\ \textsc{Env Extend}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ok}}{\Gamma \vdash_\Sigma n : \mathsf{float}}\ \textsc{Const} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad (x:T) \in \Gamma}{\Gamma \vdash_\Sigma x : T}\ \textsc{Var} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad (a:T) \in \Gamma}{\Gamma \vdash_\Sigma a : T}\ \textsc{Name}$$

$$\frac{\Gamma, K \vdash_\Sigma e : T : FS_1 \quad x \notin dom(\Gamma) \quad (\Gamma \cup \{(x:T)\}), K \vdash_\Sigma s : FS_2}{\Gamma, K \vdash_\Sigma (\mathsf{var}\ x = e; s) : (FS_1; FS_2)}\ \textsc{Bind}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ok}}{\Gamma, K \vdash_\Sigma \mathsf{skip} : \{\}}\ \textsc{Skip} \qquad \frac{\Gamma \vdash_\Sigma v : \mathsf{float} \quad \Gamma, K \vdash_\Sigma s_1 : FS \quad \Gamma, K \vdash_\Sigma s_2 : FS}{\Gamma, K \vdash_\Sigma (\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2) : FS}\ \textsc{If}$$

$$\frac{\Gamma \vdash_\Sigma v : \mathsf{float} \quad \Gamma, K \vdash_\Sigma s : FS}{\Gamma, K \vdash_\Sigma (\mathsf{while}\ (v)\ s) : FS^*}\ \textsc{While} \qquad \frac{\Gamma, K \vdash_\Sigma s : FS}{\Gamma, K \vdash_\Sigma (\mathsf{loop}\ s) : FS^\omega}\ \textsc{Loop}$$

$$\frac{\Gamma, K \vdash_\Sigma s : FS}{\Gamma, \{\} \vdash_\Sigma (\mathsf{fire}_K\ s) : \{FS \mid K\}}\ \textsc{Fire}$$

$$\frac{(f : \overrightarrow{\mathsf{float}} \to \mathsf{float}) \in \Sigma \quad \overrightarrow{\Gamma \vdash_\Sigma v_k : \mathsf{float}}}{\Gamma, K \vdash_\Sigma f(v_1, \ldots, v_k) : \mathsf{float} : \{\}}\ \textsc{Builtin}$$

$$\frac{v_1 \in \{x, l\} \quad \Gamma \vdash_\Sigma v_1 : \mathsf{float} \quad \Gamma \vdash_\Sigma v_2 : \mathsf{float}}{\Gamma, K \vdash_\Sigma v_1 = v_2 : \mathsf{float} : \{\}}\ \textsc{Assign}$$

$$\frac{(c : \mathsf{channel}\ \pi) \in \Gamma \quad \pi \leq - \quad \Gamma \vdash_\Sigma v : \mathsf{float}}{\Gamma, K \vdash_\Sigma c \uparrow v : \mathsf{float} : \{1 \cdot c\}}\ \textsc{Send} \qquad \frac{(c : \mathsf{channel}\ \pi) \in \Gamma \quad \pi \leq +}{\Gamma, K \vdash_\Sigma c \downarrow : \mathsf{float} : \{1 \cdot c\}}\ \textsc{Receive}$$

$$\frac{\Gamma \vdash_\Sigma v : \mathsf{actsig}(\{\}, FS)}{\Gamma, K \vdash_\Sigma \mathsf{run}\ v : \{\}}\ \textsc{Run} \qquad \frac{\Gamma, K \vdash_\Sigma s_1 : FS_1 \quad \Gamma, K \vdash_\Sigma s_2 : FS_2}{\Gamma, K \vdash_\Sigma (s_1; s_2) : (FS_1; FS_2)}\ \textsc{Seq}$$

$$\frac{\Gamma, K \vdash_\Sigma s : FS_0 \quad \Gamma \vdash_\Sigma FS \quad \Gamma, K \vdash_\Sigma FS_0 \cong FS}{\Gamma, K \vdash_\Sigma s : FS}\ \textsc{Stmt Eq}$$

Fig. 7. Type system

The next three definitions are for defining actors: the definition of an atomic actor, and operations for connecting actors on complementary open channels, with and without a delay. An atomic actor has a causality set, open channel set and flowstate specification, as with actor signatures. In addition, the actor has a (single-threaded) actor body, an expression that is constrained by the flowstate specification. The final definition returns no values, but starts the asynchronous execution of an actor that has no remaining open channels.

To describe a type system for this simple minilanguage, we add a type environment Γ, described as follows:

$$\Gamma ::= \{\} \mid \Gamma_1 \cup \Gamma_2 \mid \{(a:T)\} \mid \{(x:T)\}$$

$$\frac{\Gamma_0 = \{(c:T) \mid ((\mathbf{c},c):T) \in O\} \quad \Gamma_0, K_0 \vdash_\Sigma s : FS}{\Gamma, K \vdash_\Sigma \mathsf{actor}(O,s) : \mathsf{actsig}(O,FS) : \{\}} \textsc{Actor}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma v_i : \mathsf{actsig}(O_i, \{ES_i \mid K\}^\omega) \\ dom(O_1) \cap dom(O_2) = \{\} \quad edom(O_1) \cap edom(O_2) = \{\} \quad ((\mathbf{c}_i, c_i) : T_i) \in O_i \\ m = \mid ES_1 \mid_{c_1}, n = \mid ES_2 \mid_{c_2}, j \cdot m = k \cdot n = lcm(m,n) \\ K = (K_1 \cup K_2) \setminus \{c_1, c_2\} \quad O = (O_1 \cup O_2) \setminus \{c_1, c_2\} \quad \pi_1 = +, \pi_2 = - \\ FS = \{((j \cdot ES_1 \uplus k \cdot ES_2) \setminus \{c_1, c_2\}) \mid K\}^\omega\end{array}}{\Gamma, K \vdash_\Sigma \mathsf{connect}_{m,n}(v_1.\mathbf{c}_1, v_2.\mathbf{c}_2) : \mathsf{actsig}(O,FS) : \{\}} \textsc{Conn}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma v : \mathsf{actsig}(O, \{ES \mid K\}^\omega) \quad ((\mathbf{c}_1, c_1) : T_1), ((\mathbf{c}_2, c_2) : T_2) \in O \\ K_0 = K \setminus \{c_1, c_2\} \quad \mid ES \mid_{c_1} = m = \mid ES \mid_{c_2} \quad \pi_1 = +, \pi_2 = - \\ ES_0 = ES \setminus \{c_1, c_2\} \quad O_0 = O \setminus \{c_1, c_2\} \quad K, \Gamma \not\vdash_\Sigma c_1 < c_2\end{array}}{\Gamma, K \vdash_\Sigma \mathsf{connectSelf}_m(v.\mathbf{c}_1, v.\mathbf{c}_2) : \mathsf{actsig}(O_0, \{ES_0 \mid K_0\}^\omega) : \{\}} \textsc{Conn Self}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma v : \mathsf{actsig}(O, \{ES \mid K\}^\omega) \quad ((\mathbf{c}_1, c_1) : T_1), ((\mathbf{c}_2, c_2) : T_2) \in O \\ K_0 = K \setminus \{c_1, c_2\} \quad \mid ES \mid_{c_1} = m = \mid ES \mid_{c_2} \quad \pi_1 = +, \pi_2 = - \\ ES_0 = ES \setminus \{c_1, c_2\} \quad O_0 = O \setminus \{c_1, c_2\} \quad K, \Gamma \vdash_\Sigma c_1 < c_2\end{array}}{\Gamma, K \vdash_\Sigma \mathsf{connectSelfDelay}_m(v.\mathbf{c}_1, v.\mathbf{c}_2) : \mathsf{actsig}(O_0, \{ES_0 \mid K_0\}^\omega) : \{\}} \textsc{Conn Delay}$$

Fig. 8. Actor type rules

The type environment is not treated linearly, since we are not tracking usage of resources. We instead rely on matching a statement against a flowstate during type-checking. We also rely on some other meta-notations:

1. The expression $O \setminus V$ denotes the removal of all bindings for channel names in V from O:
$$O \setminus V = \{((\mathbf{c},c) : T) \in O \mid c \notin V\}.$$
We sometimes denote $O \setminus \{c\}$ by $O \setminus c$.
2. For a set of causalities K, we denote the removal of all constraints involving channels in V by $K \setminus V$. In other words, $K \setminus V = \{(c_1 < c_2) \in K \mid V \cap \{c_1, c_2\} = \{\}\}$.
3. For event states, we denote a multiset by the set of elements with their multiplicities, so $ES = \{m_1 \cdot a_1, \ldots, m_k \cdot a_k\}$ contains m_i occurrences of a_i (assuming $a_i \notin \{a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k\}$). Denote the number of occurrences of a_i in ES by $\mid ES \mid_{a_i} = m_i$, and say that $c \in ES$ if and only if $\mid ES \mid_c > 0$. The disjoint union $ES_1 \uplus ES_2$ adds the multiplicities of common elements, so $\mid ES_1 \uplus ES_2 \mid_c = \mid ES_1 \mid_c + \mid ES_2 \mid_c$. The expression $ES \setminus \{n \cdot c\}$ denotes the removal of n of occurrences of c from the multiset ES, so $\mid ES \setminus \{n \cdot c\} \mid_c = max(\mid ES \mid_c - n, 0)$. The expression $n \cdot ES$ denotes the multiplication of the multiplicities in ES by n: $n \cdot ES = \{n \cdot m \cdot a \mid m \cdot a \in ES\}$.
4. Finally we denote the projection of an event state onto the names that are in a set of variables (typically the domain of a type environment) by:
$$ES[\![V]\!] = \{(m \cdot c) \in ES \mid c \in V\}.$$
The homomorphic extension of this to the projection of a flowstate is denoted by $FS[\![V]\!]$.

$$\pi \leq \pi \quad \pm \leq + \quad \pm \leq -$$

$$\frac{\Gamma \vdash K \ \mathsf{ok} \quad (c_1 < c_2) \in K}{\Gamma, K \vdash_\Sigma c_1 < c_2} \ \textsc{Caus Hyp}$$

$$\frac{\Gamma, K \vdash_\Sigma c_1 < c_2 \quad \Gamma, K \vdash_\Sigma c_2 < c_3}{\Gamma, K \vdash_\Sigma c_1 < c_3} \ \textsc{Caus Trans}$$

Fig. 9. Subtyping and subflow rules

The type system is formulated using judgements of the following forms:

$\vdash_\Sigma \Gamma \ \mathsf{ok}$	Environment
$\Gamma \vdash_\Sigma T$	Type
$\Gamma \vdash_\Sigma K$	Causal Set
$\Gamma, K \vdash_\Sigma a < b$	Causality
$\Gamma \vdash_\Sigma v : T$	Value
$\Gamma, K \vdash_\Sigma e : T : FS$	Expression
$\Gamma, K \vdash_\Sigma s : FS$	Statement

The main type rules are provided in Fig. 7. The CONST, VAR and NAME rules are used to type check values; variables and names should be bound in the environment Γ. The skip construct has empty effect, while the conditional is required to have the same effect in both branches of the conditional. There are two looping constructs. The default rule, for while loops, has an iteration type FS^*. This may not be sufficient for some circumstances, in particular for the top-level loop of an actor that is required to always offer the specified behavior (after a complete firing). Therefore we provide an additional loop construct, to separate loops in the type system that are guaranteed to never terminate. While our type system sometimes requires infinite loops, it does not attempt to prevent infinite loops, so it is possible for an actor network to fail to make observable progress because an actor is stuck in an internal loop. The progress result for the operational semantics guarantees that the network can always make progress, even if this progress is just the iteration of an infinite loop without observable behavior.

For tracking flowstate, the key rule is the BIND rule, which evaluates an expression e, and binds a new variable x to the result of this evaluation when evaluating the statement s. Since the expression may include message sending or receiving, it has a flowstate that is combined with the flowstate of the statement continuation.

The various forms of expressions are typed by the remaining rules in Fig. 7. The BUILTIN rule type-checks a arithmetic expression resulting from the application of a built-in function. The ASSIGN rule type-checks an assignment expression, which again is restricted to primitive values of base type (i.e., float). The SEND and RECEIVE rules type-check the message sending and receiving operations. The channel must have the appropriate polarity in the environment, − for sending and + for receiving. The RUN rule runs a dataflow network for which all open channels have been resolved. The return value is a dummy value. The dataflow network operates asynchronously with the parent network.

The STMT EQ rule allows the flowstate for a statement to be replaced with a flowstate that is provably equal to it. We define the notion of equality between flowstates in the next section, defining it as a bisimulation between flowstate computations.

The difficult rules are those for connecting actors together into composite actors. These rules are provided in Fig. 8, where the rules for actor expressions are provided. The ACTOR rule type checks an atomic actor expression, checking the body of the actor in an environment binding the open channels with the appropriate polarities. The CONN rule handles the case where two different actors are being connected. As we have seen, it is because of this rule, and the difficulties with aliasing, that we do not allow actor references to be copied in this language. The rule requires that the open channel names in the two actors are distinct. In practice it would obviously be useful to have a way to rename these when necessary, but it is not essential for the current account. The top-level flowstate is required to be an infinite loop type for each actor, and the new flowstate results from a merging of these infinite loop types, removing all references to the open channels on which the actors are being linked. These channel names are also removed from the open channel set for the resulting combination.

The CONN SELF and CONN DELAY rules handle the linking of two open channels in the same actor. The first of these handles the case where the addition of the new binding does not introduce a feedback loop, as reflected by the causality rules that require the data paths in elided channels within the actor. The second of these two rules handles the case where the linking would result in a feedback loop, and therefore fills the buffer for the private channel linking the open channels to introduce a delay in firing.

Our language is essentially synchronous data flow, however its generalization to cyclostatic is straightforward. We restrict the flowstate of an actor to have the form $\{FS \mid K\}^{\omega}$. To extend this to cyclostatic behavior, we generalize the form of the body $(\{FS_1 \mid K_1\}; \ldots; \{FS_m \mid K_m\})^{\omega}$. The main complication is in the type rules for connecting actors, and we eschew the details in this account.

5 Semantics

In this section, we consider an operational semantics for the language described in the previous section. We provide a heap-based semantics that binds three kinds of values on the heap:

1. Values n of base type, i.e., of type float. Every variable of type float is assumed to be mutable, therefore we bind such variables to locations l that point to their heap binding.
2. Actor values A, which may be either atomic or composite, resulting from allocating atomic actors and then connecting them together on open channels. We extend actors with *shared channel bindings* S, containing bindings of the form $(c : (k, m, T))$. An atomic actor is a special case of a composite actor:

$$\mathsf{actor}(O, s) \equiv \mathsf{actor}(O, \{\}, s)$$

That is, an atomic actor has no shared channels, only open channels, and a single thread.

$$
\begin{array}{rcl}
T \in \text{Type} & ::= & [T]_k \\
S \in \text{Shared Channels} & ::= & \{\} \mid \{(c : (k,n,T))\} \mid S \cup S \\
H \in \text{Heap} & ::= & \varepsilon \mid l \mapsto n \mid a \mapsto A \mid c \mapsto B \mid H_1 \uplus H_2 \\
HT \in \text{Heap Type} & ::= & \varepsilon \mid l : \mathsf{float} \mid a : AS \mid c : \mathsf{channel}\ \pi \mid HT_1 \uplus HT_2 \\
A \in \text{Actor} & ::= & \mathsf{actor}(O,S,P) \\
P \in \text{Process} & ::= & \mathsf{stop} \mid FS \mid (P_1 \mid P_2) \\
B \in \text{Buffer} & ::= & \varepsilon_k \mid [v]_k \mid B_1 @_k B_2 \\
C \in \text{Configuration} & ::= & (e,H) \mid (s,H) \mid (P,H)
\end{array}
$$

Fig. 10. Syntax of configurations

3. Message buffers B, which hold the values transmitted between actors on shared channels. A message buffer is simply a sequence, ensuring FIFO delivery, where $\_ @_k \_$ is the operation for appending buffers. We assume that buffers have bounded size, provided by a parameter k in the constructors and in the buffer type; the constructor operations are undefined for the case where the resulting buffer is larger than the maximum size. We denote the number of items in a buffer by $\mid B \mid$, and the maximum size of a buffer by $size(B)$. We write $[v_1, v_2, \ldots, v_m]_k$ as an abbreviation for $[v_1]_k @_k [v_2]_k @_k \ldots @_k [v_m]_k$, where $m \leq k$. We use $v ::_k B$ to denote $[v]_k @_k B$. We use $[T]_k$ to denote the type of a buffer that contains values of type T. These buffer types are not first class, since buffers are handled by the compiler. Furthermore for simplicity we restrict the contents of buffers to be floating point values, so $T = \mathsf{float}$ for any buffer type $[T]_k$.

To describe the operational semantics, we generalize the form of an actor, as described in Fig. 10. An open channel always has polarity of + or −, reflecting the fact that it is a uniplex channel. When two such shared channels are bound to the endpoints of a shared channel, the latter has polarity $\pm$. For open channels, we use the channel names as representatives of events, since an open channel is either input or output, but not both. When we instantiate open channels to shared channels, we need to distinguish the sending and receiving ends of the shared channel, for dependency checking purposes, since we need to distinguish sending and receiving events on that channel. We assume a naming convention where, for any shared channel c, there are distinguished names c^+ and c^- for the receiving and sending parts, respectively, of that channel. The receiving end c^+ of a shared channel has polarity +, while the sending end c^- has polarity − (while the channel c itself has polarity $\pm$). Whereas open channels have environment bindings of the form $(c : \mathsf{channel}\ +)$ and $(c : \mathsf{channel}\ -)$, a shared channel binding has the form $(c : (k,n,\mathsf{channel}\ \pm))$, that also implicitly binds the names c^+ and c^- for the two ends of the channel. The parameter k denotes the maximum size required of the corresponding buffer, while the parameter n denotes that a delay must be introduced in the channel for a buffer, by initializing that buffer with n default values (either $n = k$ or $n = 0$). We denote the operation of adjusting these parameters, as a result of connecting two actors and adjusting their firing rates, by:

$$j \cdot S = \{(c : (j \cdot k, j \cdot n, T)) \mid (c : (k, n, T)) \in S\}.$$

In addition to the set of open channels O in an actor interface, we now also have a set of shared channels S. We associate with each shared channel in S a number that is the number of initial values to be inserted into a buffer when it is created. When this value is non-zero, the buffer is required to provide a delay in the firing semantics.

The other extension to an actor expression is that the body is generalized from a single thread to multiple threads, arising from the linking of actors together into a composite actor or dataflow network. The body of a composite actor is the parallel composition of a collection of single-threaded actor bodies, where each body has the flowstate specification given by the original atomic actor expression. The one change is that open channels in the original flowstate constraint will have been replaced by an endpoint of a shared channel, as a result of connecting that actor with another actor. By the time a dataflow network runs, all channel names in flowstates will have been replaced by shared channel endpoints. A configuration of the operational semantics is simply a parallel composition of threads paired with a global heap. This heap is actually partitioned between the different dataflow nets that are running.

In order to reason about correctness, we define typing relations for heaps and processes, using judgements of the form:

$$\begin{array}{ll} \Gamma \vdash_\Sigma H : HT & \text{Heap} \\ \Gamma \vdash_\Sigma P : FS & \text{Process} \\ \Gamma \vdash_\Sigma B : [T]_k & \text{Buffer} \\ \Gamma \vdash_\Sigma A : AS & \text{Actor} \end{array}$$

This last judgement just checks for well-formedness of the channel flow constraints in an actor. The channels named in the constraints should be bound in the type environment (obtained from the open channels in the case of an actor signature, and from the open and shared channels in the case of an actor expression). The type rules are provided in Fig. 11. In the ACTOR rule for typing actor bodies, the closure operation $\mathscr{C}(K)$ forms the transitive closure of a set of causality constraints.

For evaluating expressions, mutable base type variables are bound to locations l, and these must be dereferenced. This dereferencing is performed by the operation of applying the heap to a value, $H(v)$, defined by:

$$\begin{aligned} H(l) &= n \text{ if } l \mapsto n \in H \\ H(a) &= A \text{ if } a \mapsto A \in H \\ H(c) &= B \text{ if } c \mapsto B \in H \\ H(v) &= v \text{ otherwise} \end{aligned}$$

For built-in functions, we assume a family of total functions $\{eval_f\}_f$ for the functions defined in the signature Σ.

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathsf{stop} : \{\}}\ \textsc{Proc Stop}$$

$$\frac{\Gamma, \{\} \vdash_\Sigma s : FS}{\Gamma \vdash_\Sigma s : FS}\ \textsc{Proc Stmt}$$

$$\frac{\Gamma \vdash_\Sigma P_1 : FS_1 \quad \Gamma \vdash_\Sigma P_2 : FS_2}{\Gamma \vdash_\Sigma (P_1 \mid P_2) : FS_1 \parallel FS_2}\ \textsc{Proc Par}$$

$$\frac{K' \supseteq K \cap fn(FS) \quad \Gamma \vdash_\Sigma P : \{FS \mid K\}}{\Gamma \vdash_\Sigma P : \{FS \mid K\}}\ \textsc{Proc Cause}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ok}}{\Gamma \vdash_\Sigma \varepsilon_k : [T]_k}\ \textsc{Buff Empty}$$

$$\frac{\Gamma \vdash_\Sigma v : T}{\Gamma \vdash_\Sigma [v]_k : [T]_k}\ \textsc{Buff Cell}$$

$$\frac{\Gamma \vdash_\Sigma B_1 : [T]_k \quad \Gamma \vdash_\Sigma B_2 : [T]_k}{\Gamma \vdash_\Sigma B_1 @_k B_2 : [T]_k}\ \textsc{Buff Join}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ok}}{\Gamma \vdash_\Sigma \varepsilon : \varepsilon}\ \textsc{Heap Empty}$$

$$\frac{\Gamma \vdash_\Sigma n : \mathsf{float}}{\Gamma \vdash_\Sigma (l \mapsto n) : (l : \mathsf{float})}\ \textsc{Heap Float}$$

$$\frac{T = \mathsf{float} \quad \Gamma \vdash_\Sigma B : [T]_k}{\Gamma \vdash_\Sigma (c \mapsto B) : (c : [T]_k)}\ \textsc{Heap Buffer}$$

$$\frac{\Gamma \vdash_\Sigma A : AS}{\Gamma \vdash_\Sigma (a \mapsto A) : (a : AS)}\ \textsc{Heap Actor}$$

$$\frac{\Gamma \vdash_\Sigma H_1 : HT_1 \quad \Gamma \vdash_\Sigma H_2 : HT_2}{\Gamma \vdash_\Sigma H_1 \uplus H_2 : HT_1 \uplus HT_2}\ \textsc{Heap Join}$$

$$\frac{\begin{array}{c}\Gamma_0 = \{(c : T) \mid ((\mathbf{c}, c) : T) \in O\} \quad \Gamma_1 = \{(c : T) \mid (c : (k, n, T)) \in S\} \\ dom(\Gamma_0) \cap dom(\Gamma_1) = \{\} \quad \Gamma_0 \cup \Gamma_1 \vdash_\Sigma P : FS \quad FS_0 = FS[\![dom(\Gamma_0)]\!]\end{array}}{\Gamma \vdash_\Sigma \mathsf{actor}(O, S, P) : \mathsf{actsig}(O, FS_0)}\ \textsc{Actor}$$

Fig. 11. Type Rules for Heaps and Processes

$$(\mathsf{stop} \mid P) \equiv P \qquad (P_1 \mid P_2) \equiv (P_2 \mid P_1) \qquad (P_1 \mid (P_2 \mid P_3)) \equiv ((P_1 \mid P_2) \mid P_3)$$

$$(\mathsf{skip}; s) \equiv s \qquad (s; \mathsf{skip}) \equiv s \qquad (s_1; (s_2; s_3)) \equiv ((s_1; s_2); s_3)$$

Fig. 12. Structural equivalence

The semantics is defined using a collection of reduction relations:

Reduction of expressions: $(e_1, H_1) \longrightarrow (e_2, H_2)$ and $(e_1, H_1) \xrightarrow{a} (e_2, H_2)$
Reduction of statements: $(s_1, H_1) \longrightarrow (s_2, H_2)$ and $(s_1, H_1) \xrightarrow{a} (s_2, H_2)$
Reduction of processes: $(P_1, H_1) \longrightarrow (P_2, H_2)$ and $(P_1, H_1) \xrightarrow{a} (P_2, H_2)$
Reduction of types: $K \vdash FS_1 \xrightarrow{a} FS_2$

$$\frac{v = eval_f(H(v_1),\ldots,H(v_k))}{(f(v_1,\ldots,v_k),H) \longrightarrow (v,H)}\ \textsc{Builtin} \qquad \frac{n = H(v) \quad H' = H[l \mapsto n]}{(l = v,H) \longrightarrow (n,H')}\ \textsc{Assign}$$

$$\frac{n = H(v) \quad k = size(H(c)) \quad |\,H(c)\,| < k \quad H' = H[c \mapsto H(c)@_k[n]_k]}{(c^- \uparrow v,H) \xrightarrow{c^-} (0,H')}\ \textsc{Send}$$

$$\frac{H(c) = [n]_k@_kB \quad H' = H[c \mapsto B]}{(c^+ \downarrow,H) \xrightarrow{c^+} (n,H')}\ \textsc{Receive}$$

$$\frac{(e,H) \xrightarrow{[a]} (n,H') \quad l \notin dom(H) \quad H' = H \cup \{l \mapsto n\}}{((\mathsf{var}\ x = e;s),H) \xrightarrow{[a]} (\{l/x\}s,H')}\ \textsc{Bind}$$

$$\frac{H(v) \neq 0}{((\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2),H) \longrightarrow (s_1,H)}\ \textsc{If True} \qquad \frac{H(v) = 0}{((\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2),H) \longrightarrow (s_2,H)}\ \textsc{If False}$$

$$\frac{(s,H) \xrightarrow{a} (s',H)}{((\mathsf{fire}_K\ s),H) \xrightarrow{a} ((\mathsf{fire}_K\ s'),H)}\ \textsc{Fire}$$

$$\frac{H(v) \neq 0}{((\mathsf{while}\ (v)\ s),H) \longrightarrow ((s;\mathsf{while}\ (v)\ s),H)}\ \textsc{While True}$$

$$\frac{H(v) = 0}{((\mathsf{while}\ (v)\ s),H) \longrightarrow (\mathsf{skip},H)}\ \textsc{While False}$$

$$\frac{(s_1,H) \xrightarrow{[a]} (s_1',H')}{((s_1;s_2),H) \xrightarrow{[a]} ((s_1';s_2),H')}\ \textsc{Seq} \qquad \frac{(P_1,H) \xrightarrow{[a]} (P_1',H')}{((P_1 \mid P_2),H) \xrightarrow{[a]} ((P_1' \mid P_2),H')}\ \textsc{Par}$$

$$\frac{\begin{array}{c}H(v) = \mathsf{actor}(\{\},S,P) \qquad dom(H) \cap \{c \mid (c : (n,T,\in))S\} = \{\} \\ H_2 = H_1 \cup \{c \mapsto dbuf_k(n) \mid (c : (n,T,\in))S\}\end{array}}{((\mathsf{run}\ v;s),H_1) \longrightarrow ((s \mid P),H_2)}\ \textsc{Run}$$

Fig. 13. Operational Semantics

The first three pairs relations describe reductions between configurations of an expression, statement and process, respectively, coupled with a heap. The heap is both input into, and output from, each reduction step. A reduction of expressions of the form $(e_1,H_1) \longrightarrow (e_2,H_2)$ denotes an internal reduction, while a reduction of the form $(e_1,H_1) \xrightarrow{a} (e_2,H_2)$ denotes a reduction that involves a communication event a. We write $(e_1,H_1) \xrightarrow{[a]} (e_2,H_2)$ to generically denote a reduction that may be either internal or involve a communication event. Similar remarks hold for reduction of statements and of processes.

$$\frac{a \notin dom(H) \quad e = \mathsf{actor}(O,s) \quad H' = H \cup \{a \mapsto \mathsf{actor}(O,\{\},s)\}}{((\mathsf{var}\ x = e; s), H) \longrightarrow (\{a/x\}s, H')}\ \textsc{Actor}$$

$$\frac{\begin{array}{c} H(a_i) = \mathsf{actor}(O_i,S_i,P_i) \quad ((\mathbf{c}_i,c_i) : T_i) \in O_i \quad O = (O_1 \cup O_2) \setminus \{c_1,c_2\} \\ \pi_1 = +, \pi_2 = - \quad j_1 \cdot m = j_2 \cdot n = lcm(m,n) = k \quad dom(S_1) \cap dom(S_2) = \{\} \\ c \notin dom(S_1) \cup dom(S_2) \quad S = j_1 \cdot S_1 \cup j_2 \cdot S_2 \cup \{(c : (k,0,T))\} \\ P_1' = \{c^+/c_1\}P_1 \quad P_2' = \{c^-/c_2\}P_2 \quad A = \mathsf{actor}(O,S,(P_1' \mid P_2')) \end{array}}{((\mathsf{var}\ x = \mathsf{connect}_{m,n}(a_1.c_1, a_2.c_2); s), H) \longrightarrow (\{a/x\}s, H \cup \{a \mapsto A\})}\ \textsc{Conn}$$

$$\frac{\begin{array}{c} H(a) = \mathsf{actor}(O,S,P) \quad ((\mathbf{c}_1,c_1) : T_1), ((\mathbf{c}_2,c_2) : T_2) \in O \\ dom(S_1) \cap dom(S_2) = \{\} \quad c \notin dom(S_1) \cup dom(S_2) \\ S' = S \cup \{(c : (m,m,T))\} \quad \pi_1 = +, \pi_2 = - \quad O' = O \setminus \{c_1,c_2\} \\ P' = \{c^+/c_1, c^-/c_2\}P \quad A = \mathsf{actor}(O',S',P') \end{array}}{((\mathsf{var}\ x = \mathsf{connectSelfDelay}_m(a.c_1, a.c_2); s), H) \longrightarrow (\{a/x\}s, H \cup \{a \mapsto A\})}\ \textsc{Conn Delay}$$

Fig. 14. Actor operational semantics

The reduction relation for flowstates is perhaps surprising, and reflects the use of flowstate: Types themselves evolve during computation, since they are abstract process descriptions for the underlying sequential program. This reduction relation is the basis for a type equality for types, based on bisimulation:

Definition 1 (Flowstate Equality). *Given a causality set K. Define K-*bisimilarity *to be the largest symmetric binary relation R defined by: If $(FS_1, FS_2) \in R$, then if $K \vdash FS_1 \xrightarrow{a} FS_1'$ for some FS_1', then $K \vdash FS_2 \xrightarrow{a} FS_2'$ for some FS_2' such that $(FS_1', FS_2') \in R$. If FS_1 and FS_2 are K-bisimilar, then we denote this by $\Gamma, K \vdash_\Sigma FS_1 \cong FS_2$.*

The RUN rule for launching a dataflow net initializes buffers using the function $dbuf_k(n)$, defined by:

$$dbuf_k(0) = \varepsilon_k$$
$$dbuf_k(n+1) = 0 ::_k dbuf_k(n)$$

The operational semantics for agent expressions are provided in Fig. 14. The reduction relation for flowstates is defined in Fig. 15.

The formal results for the type system are in two parts:

1. *Type preservation* verifies that the well-typedness (but not the types!) of actors are preserved under evaluation.
2. *Progress* verifies that, given a flowstate that can perform a reduction step, a corresponding actor with that flowstate either diverges (loops infinitely) or eventually (after internal reductions) can simulate that abstract reduction step.

Theorem 1 (Type Preservation). *If $\Gamma, K \vdash_\Sigma (P_1, H_1) : FS_1$ and $(P_1, H_1) \xrightarrow{a} (P_2, H_2)$ then $K \vdash FS_1 \xrightarrow{a} FS_2$, and $\Gamma, K \vdash_\Sigma (P_2, H_2) : FS_2$.*

$$\{\}^* \equiv \{\} \qquad FS^\omega \equiv FS; FS^* \qquad FS_1 \parallel FS_2 \equiv FS_2 \parallel FS_1$$

$$(FS_1; FS_2); FS_3 \equiv FS_1; (FS_2; FS_3) \qquad (FS_1 \parallel FS_2) \parallel FS_3 \equiv FS_1 \parallel (FS_2 \parallel FS_3)$$

$$FS_1^\omega \parallel FS_2^\omega \equiv (FS_1 \parallel FS_2)^\omega$$

$$\{FS_1 \mid K\} \parallel \{FS_1 \mid K\} \equiv \{(FS_1 \parallel FS_2) \mid K\}$$

$$\frac{K \vdash FS_1 \xrightarrow{a} FS_1'}{K \vdash (FS_1 \parallel FS_2) \xrightarrow{a} (FS_1' \parallel FS_2)}$$

$$\frac{K_0 \cup K \vdash FS \xrightarrow{a} FS'}{K_0 \vdash \{FS \mid K\} \xrightarrow{a} \{FS \mid K\}}$$

$$\frac{K \vdash FS_1 \xrightarrow{a} FS_1'}{K \vdash (FS_1; FS_2) \xrightarrow{a} (FS_1'; FS_2)}$$

$$\frac{K \vdash FS \xrightarrow{a} FS'}{K \vdash (\{\}; FS) \xrightarrow{a} (\{\}; FS)}$$

$$\frac{m > 0 \qquad \nexists a_0 \in ES.\Gamma, K \vdash_\Sigma a_0 < a}{K \vdash \{m \cdot a\} \uplus ES \xrightarrow{a} \{(m-1) \cdot a\} \uplus ES}$$

$$\frac{FS_1 \equiv FS_1' \qquad K \vdash FS_1' \xrightarrow{a} FS_2' \qquad FS_2 \equiv FS_2'}{K \vdash FS_1 \xrightarrow{a} FS_2}$$

Fig. 15. Type reduction rules

Let $(P,H) \Longrightarrow (P',H')$ denote the reflexive transitive closure of $(P,H) \longrightarrow (P',H')$: in other words, (P,H) evolves to (P',H') in zero or more internal reductions. Let $(P_1,H_1) \stackrel{a}{\Longrightarrow} (P_2,H_2)$ denote that $(P_1,H_1) \Longrightarrow (P_1',H_1')$ and $(P_1',H_1') \xrightarrow{a} (P_2',H_2')$ and $(P_2',H_2') \Longrightarrow (P_2,H_2)$. Let $(P_1,H_1) \stackrel{(a_1,\ldots,a_k)}{\Longrightarrow} (P_{k+1},H_{k+1})$ denote that $(P_i,H_i) \stackrel{a_i}{\Longrightarrow} (P_{i+1},H_{i+1})$ for $i = 1,\ldots,k$ and some $(P_1,H_1),\ldots,(P_{k+1},H_{k+1})$ and $a_1,\ldots,a_k$.

Denote that a configuration (P,H) *diverges*, in the sense that it loops indefinitely performing only internal reductions, by $(P,H) \Uparrow$. Denote that a configuration eventually offers output on channel endpoint c^-, perhaps after performing some internal reductions, by $(P,H) \Downarrow_{c^-}$. Similarly $(P,H) \Downarrow_{c^+}$ denotes that a configuration eventually attempts to perform input on channel endpoint c^+, perhaps after performing some internal reductions.

Theorem 2 (Progress). *If* $\Gamma, K \vdash_\Sigma (P,H) : FS_1$ *and* $\left\{ \begin{array}{l} K \vdash FS_1 \xrightarrow{c^+} FS_2 \\ K \vdash FS_1 \xrightarrow{c^-} FS_2 \end{array} \right\}$ *then either* $(P,H) \Uparrow$, *or* $\left\{ \begin{array}{l} (P,H) \stackrel{\overrightarrow{a^+}}{\Longrightarrow} (P',H') \\ (P,H) \stackrel{\overrightarrow{a^-}}{\Longrightarrow} (P',H') \end{array} \right\}$ *for some* (P',H') *and* $\overrightarrow{a}$ *such that* $\left\{ \begin{array}{l} (P',H') \Downarrow_{c^+} \\ (P',H') \Downarrow_{c^-} \end{array} \right\}$.

6 Related Work

The area of synchronous dataflow has seen some application in signal processing applications [13], with various extensions, in particular cyclostatic dataflow for actors whose firing behavior is able to evolve in a regular fashion [2]. Traditionally the analysis of synchronous dataflow has been non-modular, requiring analysis of the entire dataflow graph. More recent work has considered the modular composition of hierarchical SDF graphs [17], allowing graphs to be analyzed before the entire dataflow graph is constructed, with a focus on modular code generation. The interface of a modular actor is described as a deterministic SDF with shared FIFOs (DSSF) profile, allowing a collection of actors to share an input queue while retaining determinacy of the execution. As with SDF, atomic actors are considered as "black boxes," and only SDF is considered. In particular cyclostatic dataflow is not considered in that work.

Sessional dataflow comes out of the realm of linear [11,18] and affine type systems for statically checking the safe usage of limited resources. Two particularly significant lines of study in the "linear types" field have been the approach of *typestate* and that of *session types*. Typestate is a concept that originated in the Hermes language of Strom and Yemini [15]. It corresponds to an enrichment of the normal notion of a type, to include the concept of types as states in a finite state machine. Fähndrich and Deline incorporated this idea into object oriented languages [7] in a very natural way: each object has a typestate, and the interface offered by an object, in the sense of the methods that can currently be invoked on the object, are determined by its current typestate. Since typestate is updated imperatively, it is important that aliasing of such objects be carefully controlled. Aldrich et al [14] have demonstrated that a notion of *permissions*, based on earlier work on type-based capabilities, can be used to check the use of typestate in existing non-toy software systems. We have explicitly avoided introducing these issues into the current report, but they are clearly relevant to incorporating sessional dataflow into real programming languages.

The approach of session types [9] is commonly motivated by its support for safe Web services. In the simplest case, session types are used to mediate the exchanges between two parties in a dyadic interaction. Each session offers a "shared channel" (different from our use of the terminology), essentially an service endpoint URL that a client connects to. On connection, a new server thread is forked and a private session channel is established between the client and this thread. This channel has a behavioral type that is essentially an abstract single-threaded process, that constrains the communications between the parties. Since only the client and the server share their private channel, the execution is in fact deterministic.

Although sessional dataflow might appear at first related to session types, the connection is actually rather weak, because of the nature of the interactions in dataflow. The closest our system comes to a session types system is in the behavioral constraint on the behavior of an actor, in terms of matching the specified input and output data rates on each firing specified on an actor interface. However this behaviorial specification only constrains a single actor, and places no constraint on the behavior of its neighboring actors (upstream or downstream). Furthermore a session type specifies, for each participant in an interaction, a very precise single-threaded behavior, in terms of data

exchanged on the private session channels at each point in the execution. In contrast, the behavioral specification for an actor in sessional dataflow is declarative, specifying expected communications subject to causality constraints. Deniélou and Yoshida [8] describe a version of session types that allows a dynamic number of participants in a session protocol. As with other approaches to session types, the approach is to provide operational specifications of participant behaviors, using a top-down approach where one reasons from the specified global protocol to the behavior of individual participants. In contrast, the sessional dataflow approach is bottom-up and declarative, specifying declarative causality constraints on individual actors independent of whatever interactions they are integrated into.

Another related line of work is in synchronous languages for real-time and embedded systems. Such languages assume a "clock" on all computations, with variables representing potentially infinite streams of values, indexed by clock ticks. Here the most relevant example for sessional dataflow is that of Lustre [4], a language that is a dataflow language in the tradition of Lucid, [1], and is a synchronous language in the sense of the synchronous languages such as Esterel [3], but which we cannot call a synchronous dataflow language for fear of confusing the reader. The constraints on the synchronous languages preclude any need for buffering, since all actors operate in lock step on the same clock. The theory of these "synchronous," "dataflow" networks has been described in terms of synchronous Kahn networks [5], which have the property that no buffering is required at all between actors, since all execution is synchronous and governed by a common clock. This is clearly a very strong restriction, albeit one that facilitates compilation of programs to hardware circuits. The theory of N-synchronous Kahn networks [6] relaxes this restriction, allowing different actors to have their own clock rates, and allowing buffering between actors to match their clock rates. It is therefore very much related to the approach of synchronous dataflow, with subtyping between clock rates in multi-rate systems identifying where data must be buffered. However matching data rates does not address the other aspect of sessional dataflow, causalities to ensure the liveness of networks as they are composed.

7 Conclusions

This work builds on existing work in dataflow computation, particularly the work in synchronous dataflow pursued in the signal processing community, as discussed in Sect. 1. Our work considers how to relate the implementations of actors to the static firing rates described in actor interfaces, where the latter are critical for static scheduling of actors. We have also provided a compositional semantics for combining actors together into dataflow nets, in such a way that we statically check the correctness of the combination at each step of such a process.

There are extensions of synchronous dataflow that can be incorporated into sessional dataflow, such as the extension to cyclostatic dataflow considered at the end of Sect. 4. However our main interest is in using the framework of sessional dataflow to consider the safety of operations such as reconfiguration and subnet replacement.

References

1. Ashcroft, E.A., Wadge, W.W.: Lucid, the dataflow programming language. Academic Press (1985)
2. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static data flow. In: International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol. 5, pp. 3255–3258 (May 1995)
3. Boussinot, F., de Simone, R.: The Esterel language. Proc. IEEE 79, 1270–1282 (1991)
4. Boussinot, F., de Simone, R.: The synchronous data flow programming language Lustre. Proc. IEEE 79, 1305–1320 (1991)
5. Caspi, P., Pouzet, M.: Synchronous kahn networks. In: International Conference on Functional Programming, ICFP (1996)
6. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In: Principles of Programming Languages (POPL), pp. 180–193. ACM Press (2006)
7. DeLine, R., Fähndrich, M.: Typestates for Objects. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)
8. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: ACM Symposium on Principles of Programming Languages, pp. 435–446. ACM, New York (2011)
9. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and Session Types: An Overview. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 1–28. Springer, Heidelberg (2010)
10. Edwards, S.A.: Languages for Digital Embedded Systems. Kluwer (2000)
11. Girard, J.-Y.: Linear logic. Theoretical Computer Science (50), 1–102 (1987)
12. Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing 74: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, Stockholm (1974)
13. Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE 75(9), 1235–1245 (1987)
14. Stork, S., Marques, P., Aldrich, J.: Concurrency by default: using permissions to express dataflow in stateful programs. In: Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 933–940. ACM, New York (2009)
15. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. 12, 157–171 (1986)
16. Thies, W.: Language and Compiler Support for Stream Programs. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (February 2009)
17. Tripakis, S., Bui, D., Rodiers, B., Lee, E.A.: Compositionality in synchronous data flow: Modular code generation from SDF graphs. Technical Report UCB/EECS-2009-143, University of California, Berkeley (October 2009)
18. Wadler, P.: Linear types can change the world? In: Programming Concepts and Methods. North (1990)

Java Wildcards Meet Definition-Site Variance

John Altidor[1], Christoph Reichenbach[1], and Yannis Smaragdakis[1,2]

[1] University of Massachusetts, Amherst
[2] University of Athens, Greece

Abstract. Variance is concerned with the interplay of parametric polymorphism (i.e., templates, generics) and subtyping. The study of variance gives answers to the question of when an instantiation of a generic class can be a subtype of another. In this work, we combine the mechanisms of use-site variance (as in Java) and definition-site variance (as in Scala and C#) in a single type system, based on Java. This allows maximum flexibility in both the specification and use of generic types, thus increasing the reusability of code. Our VarJ calculus achieves a safe synergy of def-site and use-site variance, while supporting the full complexities of the Java realization of variance, including F-bounded polymorphism and wildcard capture. We show that the interaction of these features with definition-site variance is non-trivial and offer a full proof of soundness—the first in the literature for an approach combining variance mechanisms.

1 Introduction

Consider a generic type `C<X>`. When is a type-instantiation `C<`*Exp1*`>` a subtype of another type instantiation `C<`*Exp2*`>`? This is the question that *variance* mechanisms in modern programming languages try to answer. Variance (specifically, subtype variance with respect to generic type parameters) is a key topic in language design because it develops the exact rules governing the interplay of the two major forms of polymorphism: parametric polymorphism (i.e., generics or templates) and subtype (inclusion) polymorphism.

Languages like C# and Scala support a type system with *definition-site variance*: at the point of defining the generic type `C<X>` we state its subtyping policy and the type system attempts to prove that our assertion is statically safe. For instance, a C# definition `class C<out X> ...` means that `C` is *covariant*: `C<S>` is a subtype of `C<T>` if `S` is a subtype of `T`. The type system's obligation is to ensure that type parameter `X` of `C` is used in the body of `C` in a way that guarantees type safety under this subtyping policy. For instance, `X` cannot appear as the argument type of a public method in `C`—a rule colloquially summarized as "the argument type of a method is a contravariant position".

By contrast, the type system of Java employs the concept of *use-site variance* [11]: a class does not itself state its variance when it is defined. Uses of the class, however, can choose to specify that they are referring to a *covariant*, *contravariant*, or *bivariant* version of the class. For instance, a method `void`

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 509–534, 2012.

`meth(C<? extends T> cx)` can accept arguments of type `C<T>` but also `C<S>` where `S` is a subtype of `T`. An object with type `C<? extends T>` may not offer the full functionality of a `C<T>` object: the type system ensures that the body of method `meth` employs only such a subset of the functionality of `C<T>` that would be safe to use on any `C<S>` object (again, with `S` a subtype of `T`). This can be viewed informally as automatically projecting class `C` and deriving per-use versions.

Each flavor of variance has its own advantages. Use-site variance is arguably a more advanced idea, yet it suffers from specific usability problems because it places the burden on the *user* of a generic type. (Although one should keep in mind that the users of one generic type are often the implementors of another.) Definition-site variance may be less expressive, but leaves the burden of specifying general interfaces with the *implementor* of a generic. A natural idea, therefore, is to combine the two flavors in the same language design and allow full freedom: For instance, when a type is naturally covariant, its definition site can state this property and relieve the user from any further obligation. Conversely, when the definition site does not offer options for fully general treatment of a generic, a sophisticated user can still provide fully general signatures.

This natural combination of the two kinds of variance is complicated especially by the interaction of use-site and definition-site annotations: for example, when does the declared variance of a type variable agree with occurrences of that variable in use-site annotations? We recently proposed a unifying framework for checking and inferring both definition and use-site variance [1]. That proposal was not accompanied by a language operational semantics however—its proof of soundness was expressed as a meta-theorem, i.e., under assumptions over what an imaginary language's type system should be able to prove about sets of values. This meta-theorem was welcome as an intuition about why it makes sense to combine variances in a certain way, but did not establish a firm connection with any real programming language.

This paper investigates combining of definition- and use-site variance with all relevant language constructs using a new formal model, VarJ. VarJ applies novel ideas and integrates techniques from various formalisms: Java wildcards are a form of use-site variance that was proven sound with the TameFJ [4] calculus. VarJ directly extends TameFJ with definition-site variance. VarJ also employs ideas from our VarLang calculus [1], which introduces a variance transform operator. Finally, VarJ integrates definition-site subtyping rules from the work of Kennedy et al. [13,8]. The result is a language with highly expressive genericity. For instance, given an invariant class `List`, our type system allows defining a (definition-site) covariant class `ROStack` that returns covariant (intuitively: read-only) `List`s of members:

```
class ROStack<+X> {
  X pop() { ... }
  List<? extends X> toList() { ... }
}
```

Note the simultaneous use of a definition-site variance annotation (`+`) on `ROStack`, as well as a use-site variance annotation on its `toList` method. The former is not safe without the latter.

Overall our work makes several contributions. At the high level:

- Compared to the type systems of Java, C#, or Scala, our combination of definition-site and use-site variance allows the programmer to pick the best tool for the job. Libraries can avoid offering different flavors of interfaces just to capture the notion of, e.g., "the covariant part of a list" vs. "the contravariant part of a list". Conversely, users can often use purely-variant types more easily and with less visual clutter if the implementor of that type had the foresight to declare its variance.
- Our approach maintains other features of the Java type system, namely full support for wildcards, which are a mechanism richer than plain use-site variance (e.g., [10]) and allow uses directly inspired by existential types.
- We provide a framework for determining the variance of the various positions in which a type can occur. (For example, why is the upper bound of the type parameter of a polymorphic method a contravariant position?)

Also, at the technical level:

- We show how definition-site variance interacts in interesting ways with advanced typing features, such as existential types, F-bounded polymorphism, and wildcard capture. A naive application of our earlier work [1] to Java would result in unsoundness, as we show with concrete examples. (Our earlier approach avoided unsoundness when applied to actual Java code by making several over-conservative assumptions to completely eliminate any interaction between, e.g., definition-site variance and F-bounded polymorphism.)
- We clarify and extend the TameFJ formalism with definition-site variance. TameFJ is a thorough, highly-detailed formalism and extending it is far from a trivial undertaking. The result is that we offer the first full formal modeling and proof of soundness for a language combining definition- and use-site variance.

2 Background

We next offer a brief background on definition- and use-site variance as well as their relative advantages.

2.1 Definition-Site Variance

Languages supporting definition-site variance [14,9] typically require each type parameter to be declared with a variance annotation. For instance, Scala [14] requires the annotation `+` for covariant type parameters, `-` for contravariant type parameters, and invariance as default. A well-established set of rules can then be used to verify that the use of the type parameter in the generic[1] is consistent with the annotation.

In intuitive terms, we can understand the restrictions on the use of type parameters as applying to "positions". Each typing position in a generic's signature

[1] We refer to all generic types (e.g., classes, traits, interfaces) uniformly as "generics".

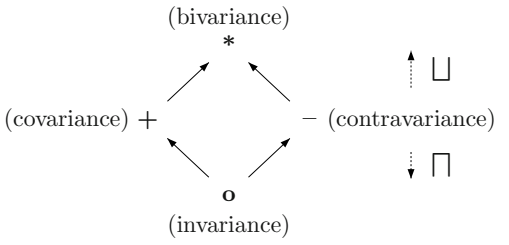


Fig. 1. Standard variance lattice

has an associated variance. For instance, method return and exception types, supertypes, and upper bounds of class type parameters are covariant positions; method argument types and class type parameter lower bounds are contravariant positions; field types are both co- and contravariant occurrences, inducing invariance. Type checking the declared variance annotation of a type parameter requires determining the variance of the positions the type parameter occurs in. *The variance of all such positions should be at least the declared variance of the type parameter.* Figure 1 presents the variance lattice. Consider the following templates of Scala classes, where v_X, v_Y, and v_Z stand for variance annotations.

```
abstract class RList[v_X X] { def get(i:Int):X }
abstract class WList[v_Y Y] { def set(i:Int, y:Y):Unit }
abstract class IList[v_Z Z] { def setAndGet(i:Int, z:Z):Z }
```

The variance v_X is the declared definition-site variance for type variable `X` of the Scala class `RList`. If $v_X = +$, the `RList` class type checks because `X` does not occur in a contravariant position. If $v_Y = +$, the `WList` class does not type check because `Y` occurs in a contravariant position (second argument type in `set` method) but $v_Y = +$ implies `Y` should only occur in covariant position. `IList` type checks only if $v_Z = o$ because `Z` occurs in both a covariant and a contravariant position.

Intuitively, `RList` is a read-only list: it only supports retrieving objects. The return type of a method indicates this "retrieval" capability. Retrieving objects of type `T` can be safely thought of as retrieving objects of any supertype of `T`. Thus, a read-only list of `T`s (`RList[T]`) can always be safely thought of as a read-only list of some supertype of `T`s (`RList[S]`, where `T<:S`). This is the exact definition of covariant subtyping and the reason why a return type is a covariant position. Thus, `RList` is covariant in `X`. Similarly, `WList` is a write-only list, and is intuitively *contravariant*. Its definition supports this intuition: Objects of type `T` can be written to a write-only list of `T`s (`WList[S]`) and written to a write-only list of `S`s (`WList[S]`), where `T<:S`, because objects of type `T` are also objects of type `S`. Hence, a `WList[S]` can safely be thought of as a `WList[T]`, if `T<:S`.

The variance of type variables is *transformed* by the variance of the context the variables appear in. Covariant positions *preserve* the variance of types that appear in them, whereas contravariant positions *reverse* the variance of the types that appear in them. The "reverse" of covariance is contravariance, and vice versa. The "reverse" of invariance is itself. Thus, we can consider the occurrence

of a type parameter to be initially covariant. For instance, consider again the Scala classes above. In `RList`, `X` only appears as the return type of a method, which preserves the initial covariance of `X`, so `RList` is covariant in `X`. In `WList`, `Y` appears in a contravariant position, which reverses its initial covariance, to contravariance. Thus, `WList` is contravariant.

When a type parameter is used to instantiate a generic, its variance is further transformed by the declared definition-site variance of that generic. For example:

```
class SourceList[+Z] { def copyTo(to:WList[Z]):Unit }
```

Suppose the declared definition-site variance of `WList` (with respect to its single parameter) is contravariance. In `WList[Z]`, the initial covariance of `Z` is transformed by the definition-site variance of `WList` (contravariance). It is then transformed again by the contravariant method argument position. As a result, `Z` appears covariantly in this context, and `SourceList` is covariant in `Z`, as declared. Any variance transformed by invariance becomes invariance. Thus, if `Z` had been used to parameterize an invariant generic, its appearance would have been invariant.

We have so far neglected to discuss *bivariance*: `C<X>` is bivariant implies that `C<S><:C<T>` for any types `S` and `T`. Declaring a bivariant type parameter is not supported by the widely used definition-site variant languages, since designating a type parameter as bivariant typically means it does not appear in the generic's type signature. Nevertheless, the concept is useful in our more general treatment.

2.2 Use-Site Variance

An alternative approach to variance is *use-site variance* [11,18,4]. Instead of declaring the variance of `X` at its definition site, generics are assumed to be *invariant* in their type parameters. However, a type-instantiation of `C<X>` can be made co-, contra-, or bivariant using variance annotations.

For instance, using the Java wildcard syntax, `C<? extends T>` is a *covariant* instantiation of `C`, representing a type "`C`-of-some-subtype-of-`T`". `C<? extends T>` is a supertype of all type-instantiations `C<S>`, or `C<? extends S>`, where `S<:T`. In exchange for such liberal subtyping rules, type `C<? extends T>` can only access fully those methods and fields of `C` in which `X` appears covariantly. (Other methods can be used only with type-neutral values, e.g., called with `null` instead of values of type `X`.) In determining this, use-site variance applies the same set of rules used in definition-site variance, with the additional condition that the upper bound of a wildcard is considered a covariant position, and the lower bound of a wildcard a contravariant position.

For example, consider an invariant generic class `List` that uses its type parameter in both covariant and contravariant positions:

```
class List<X> {
  ...  // other members that don't affect variance
  void add(int i, X x) { ... }  // requires a List<? super X>
  X get(int i) { ... }          // requires a List<? extends X>
  int size() { ... }            // requires a List<?>
}
```

`List<? extends T>`, only has access to method "`X get(int i)`", but not method "`void add(int i, X x)`". (More precisely, method `add` can only be called with `null` for its second argument.)

Similarly, `C<? super T>` is the contravariant version of `C`, and is a supertype of any `C<S>` and `C<? super S>`, where `T`<:`S`. Of course, `C<? super T>` has access only to methods and fields in which `X` appears contravariantly or not at all. (The `get` method returns `Object` for a `C<? super T>`.)

Use-site variance also allows the representation of the *bivariant* version of a generic. In Java, this is accomplished through the unbounded wildcard: `C<?>`. Using this notation, `C<S>`<:`C<?>`, for any `S`. The bivariant type, however, only has full access to methods and fields in which the type parameter does not appear at all. In definition-site variance, these methods and fields would have to be factored out into a non-generic class.

2.3 A Comparison

Both approaches to variance have their merits and shortcomings. Definition-site variance enjoys a certain degree of conceptual simplicity: the generic type instantiation rules and subtyping relationships are clear. However, the class or interface designer must pay for such simplicity by splitting the definitions of data types into co-, contra, and bivariant versions. This can be an unnatural exercise. For example, the data structures library for Scala contains immutable (covariant) and mutable (invariant) versions of almost every data type—and this is not even a complete factoring of the variants, since it does not include contravariant (write-only) versions of the data types.

The situation gets even more complex when a generic has more than one type parameter. In general, a generic with n type parameters needs 3^n (or 4^n if bivariance is allowed as an explicit annotation) interfaces to represent a complete variant factoring of its methods. Arguably, in practice, this is often not necessary.

Use-site variance, on the other hand, allows users of a generic to create co-, contra-, and bivariant versions of the generic on the fly. This flexibility allows class or interface designers to implement their data types in whatever way is natural. The users of these generics must pay the price, by carefully considering the correct use-site variance annotations, so that the type can be as general as possible. This might not seem very difficult for a simple instantiation such as `List<? extends Number>`. However, type signatures can very quickly become complicated. For instance, the following method signature is part of the Apache Commons-Collections Library:

```
Iterator<? extends Map.Entry<? extends K,V>>
  createEntrySetIterator(Iterator<? extends Map.Entry<? extends K,V>>)
```

3 Combining Definition- and Use-Site Variance

Our formalism supports combining definition- and use-site variance in the context of Java. We next see informally some of its main insights and complications.

3.1 Insights for Combining Variances

High-level elements of our approach are inherited from our earlier work [1], in which we presented rules for combining definition- and use-site variance in a type system. These rules are a significant generalization over what has been explored in the past (mainly in Scala) in two ways:[2]

- The variance of an arbitrary type expression with respect to a type variable is defined for all cases with the help of a "transform" operator, $\otimes$. The operator determines how variances compose. Given two generic types `A<X>` and `B<X>` with declared variances v_A and v_B for their parameters (i.e., declared as `A<`v_A `X>` and `B<`v_B `X>`), we can compute the variance of type `A<B<X>>` as $v = v_A \otimes v_B$. Variances take values from the lattice of Figure 1.

  Figure 2 summarizes the behavior of the transform operator: invariance and bivariance override other variances, while covariance preserves and contravariance reverses variance (with invariance and bivariance being their own reverses, respectively). To sample why the definition of the transform operator makes sense, we derive one case relating the inputs and output. (Remaining cases are derived similarly.)

  – **Case** $+ \otimes - = -$: This means that type expression `C<`E`>` is contravariant with respect to type variable `X` when generic `C` is covariant in its type parameter and type expression E is contravariant in `X`. This is true because, for any two types T_1 and T_2:

$$
\begin{aligned}
& T_1 <: T_2 \\
\implies\ & E\texttt{[}T_2\texttt{/X]} <: E\texttt{[}T_1\texttt{/X]} && \text{(by contravariance of } E\text{)} \\
\implies\ & \texttt{C<}E\texttt{[}T_2\texttt{/X]>} <: \texttt{C<}E\texttt{[}T_1\texttt{/X]>} && \text{(by covariance of C)} \\
\implies\ & \texttt{C<}E\texttt{>[}T_2\texttt{/X]} <: \texttt{C<}E\texttt{>[}T_1\texttt{/X]}
\end{aligned}
$$

  Hence, `C<E>` is contravariant with respect to `X`.

Definition of variance transformation: $\otimes$

$+ \otimes + = +$	$- \otimes + = -$	$* \otimes + = *$	$o \otimes + = o$
$+ \otimes - = -$	$- \otimes - = +$	$* \otimes - = *$	$o \otimes - = o$
$+ \otimes * = *$	$- \otimes * = *$	$* \otimes * = *$	$o \otimes * = *$
$+ \otimes o = o$	$- \otimes o = o$	$* \otimes o = *$	$o \otimes o = o$

Fig. 2. Variance transform operator

- The interaction of use-site and definition-site variance is expressed as a join operation on the same variance lattice of Figure 1. In the VarLang calculus [1], types have the form `C<`$\overline{vT}$`>`, where $\overline{v}$ are use-site annotations. Considering the `List` generic from Section 2.2, for example, `List<+T>` passes type `T` to a *covariant*

[2] There is also a third way: our earlier framework allowed reasoning about unknown variances, represented as variables in recursive constraints, thus enabling variance *inference* instead of *checking*. This capability is not relevant here, however.

version of the `List`, where the `add` method was "removed"[3] because it contains the type parameter in a contravariant position. If generic `C<X>` has definition-site variance v_1 with respect to `X`, then the type expression `C<`v_2`X>` has variance $v_1 \sqcup v_2$ with respect to `X`. Consider a covariant class `RList`. If we request a contravariant instantiation, we end up with a bivariant type expression ($+\sqcup- = *$). That is, a method "`void foo(RList<? super Animal> l) {...}` can really accept any `RList`: the method is guaranteed to never use its argument in a way that reveals anything about the type of element in the list. (In practice, this means that the method may only take the size of the list, or only treats its elements as being of the general type `Object`, etc.)

In practice, the ability to combine definition- and use-site variance gives the programmer maximum flexibility. The variance of a generic class does not need to be anticipated at its definition site. Consider the usual invariant `List` class (from Section 2.2). This `List` supports both reading and writing of data, hence it includes both kinds of methods, instead of being split into two types (as, for instance, is common in the Scala libraries). The methods that use `List` can still be made fully general, however, as long as they specify use-site annotations. Generally, allowing both kinds of variance in a single language ensures modularity: parts of the code can be made fully general regardless of how other code is defined.

At the same time, allowing definition-site variance eliminates much of the need for extensive use-site variance annotations and the risk of too-restricted types: purely variant types can be declared up-front without burdening the programmer at the use point.

Of course, combining definition- and use-site variance means more than just using each kind separately, when applicable. It also includes using one kind of annotation when reasoning about the other. For instance, consider the example of a read-only stack type that we briefly saw in the Introduction. The stack refers to the invariant class `List` (defined earlier):

```
class ROStack<+X> {
  X pop() { ... }
  List<? extends X> toList() { ... }
}
```

Note the use of a definition-site variance annotation (`+`) on `ROStack`, as well as a use-site variance annotation on its `toList` method. The example would not have been safe if the return type of `toList` were merely `List<X>`. With such an invariant use of type parameter `X`, we could use a (dynamic) `List<Dog>` as a (static) `List<Animal>` and thus dynamically add a `Cat` (which is fine to add to a `List<Animal>`) to a `List<Dog>`.

3.2 Realistic Complications

The main contribution of our present work consists of formalizing and proving sound the combination of definition- and use-site variance in the context of Java.

[3] Again, method `add` can only be called with `null` for its second argument.

In order to do so, we need to reason about the interaction of definition-site variance with many complex language features, such as F-bounded polymorphism, polymorphic methods, bounds on type parameters, and existential-types (arising in the use of wildcards). This interaction is highly non-trivial, as we see in examples next.

One complication is that Java wildcards are not merely use-site variance, but also include mechanisms inspired by existential typing mechanisms. *Wildcard capture* is the process of passing an unknown type, hidden by a wildcard, as a type parameter in a method invocation. Consider the following method, which swaps the order the two elements at the top of a stack.

```
<E> void swapLastTwo(Stack<E> stack)
{ E elem1 = stack.pop();  E elem2 = stack.pop();
  stack.push(elem2);  stack.push(elem1);  }
```

Although a programmer may want to pass an object of type `Stack<?>` as a value argument to the `swapLastTwo` method, the type parameter to pass for `E` cannot be manually specified because the type hidden by `?` cannot be named by the programmer. However, passing a `Stack<?>` type checks because Java allows the compiler to automatically generate a name for the unknown type (capture conversion) and use this name in a method invocation. Our formalism has to model wildcard capture and its interaction with definition-site variance. This handling comprises some of the more significant changes of our formalism relative to TameFJ [4].

Another major complication concerns F-bounded polymorphism. Consider the following definition:

```
interface Trouble<P extends List<P>> extends Iterator<P> {}
```

The type `Trouble<P>` extends `Iterator<P>`, which is assumed in the example to be covariant (exporting a method "`P next()`", per the Java library convention for iterators). It would stand to reason that `Trouble` is also covariant: an object of type `Trouble<P>` does precisely what an `Iterator<P>` object does, since it simply inherits methods. Consider, however, the type `Trouble<? super A>`. This is a contravariant use of a covariant generic. According to our approach for combining variances, this results in a bivariant type (due to the variance joining). For example, we can derive the following subtype relationship even though the types, `MyList` and `YourList`, are not subtype related.

`Trouble<YourList>` $<:$ `Trouble<Object>` (by covariance assumption of `Trouble`)

$<:$ `Trouble<? super Object>`

$<:$ `Trouble<? super MyList>`

The problem, however, is that the bounds of type variables (`List<P>` in this case) are preserved in the existential type representing a use of `Trouble` with wildcards. This results in unsoundness because, in F-bounded polymorphism, the bound includes the hidden type, allowing its recovery and use. We can cause

a problem with the following code (`ArrayList` is a standard implementation of the usual Java `List` interface, both invariant types):[4]

```
class MyList extends ArrayList<MyList> {}
class YourList extends ArrayList<YourList> {
  int i = 0;
  public boolean add(YourList list)
  {  System.out.println(list.i); return super.add(list); }
}
void foo(Trouble<? super MyList> itr) { itr.next().add(new MyList()); }
Trouble<YourList> preitr = ...;
foo(preitr);
```

Function `foo` type checks because `itr.next()` is guaranteed to return an unknown supertype, `X`, of `MyList` but also (due to the F-bound on `Trouble`) a subtype of `List<X>`. Thus, `X` has a method `add` (from `List`) which accepts `X` instances, and thus also accepts `MyList` instances (since `X` is a supertype of `MyList`).

The problem arises in the last line. If `Trouble<? super MyList>` were truly bivariant (as a contravariant use of a covariant generic), then that line would type check, allowing the unsound addition of a `MyList` object to a list of `YourLists`. Thus, the joining of definition- and use-site variances needs to be carefully restricted in the presence of F-bounded polymorphism. We revisit the above example more formally in Section 5.

4 VarJ

We investigate extending Java with definition-site variance by developing the VarJ calculus. VarJ's syntax found in Figure 3 is a slight extension of the TameFJ syntax. A program that type checks in TameFJ also type checks in VarJ. Significant differences are highlighted using shading. To improve readability, some syntactic categories are overloaded with multiple meta-variables. *Existential types* range over `T`, `U`, `V`, and `S`. *Type variables* range over `X`, `Y`, and `Z`. *Bounds* range over `B` and `A`. *Variances* range over `v` and `w`. The bottom type, $\bot$, is used only as a lower bound. We follow the syntactic conventions of TameFJ: all source-level type expressions are written as existential types, with an empty range for non-wildcard Java type uses and type variables written as $\exists\emptyset.\texttt{X}$; substitution is performed as usual except $[\texttt{T}/\texttt{X}]\exists\emptyset.\texttt{X} = \texttt{T}$; $\star$ is a syntactic marker designating that a method type parameter (i.e., for a polymorphic method) should be inferred. Class type parameters ($\overline{\texttt{X}}$) now have definition-site variance annotations ($\overline{\texttt{v}}$) and lower bounds ($\overline{\texttt{B}_L}$). Method type variables now have lower bounds as well. The remainder of this section focuses on semantic differences between VarJ and TameFJ and new concepts from adding definition-site variance. Sections 4.1 and 4.2 formally present notions of the variance of type expression and the variance of a type position. Section 4.3 covers subtyping with definition-site and use-site variance in VarJ. Section 4.4 discusses the updates made to allow safe interaction between wildcard capture and variant types.

[4] This example is originally due to Ross Tate.

Syntax:

$$
\begin{array}{lll}
\texttt{e} & ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} & \textit{expressions} \\
\texttt{s} & ::= \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{s}}\texttt{)} & \textit{values} \\
\texttt{v} & ::= + \mid - \mid * \mid o & \textit{variance} \\
\\
Q & ::= \texttt{class C<}\overline{\texttt{v X} \rightarrow \texttt{[}\,\texttt{B}_L\,\texttt{-B}_U\texttt{]}}\texttt{>} \lhd \texttt{N} \{\ \overline{\texttt{T f;}}\ \ \overline{M}\ \} & \textit{class declarations} \\
M & ::= \texttt{<}\overline{\texttt{X} \rightarrow \texttt{[}\,\texttt{B}_L\,\texttt{-B}_U\texttt{]}}\texttt{>}\ \texttt{T m(}\overline{\texttt{T x}}\texttt{)}\ \{\ \texttt{return e;}\ \} & \textit{method declarations} \\
\\
\texttt{N} & ::= \texttt{C<}\overline{\texttt{T}}\texttt{>} & \textit{non-variable types} \\
\texttt{R} & ::= \texttt{N} \mid \texttt{X} & \textit{non-existential types} \\
\texttt{T} & ::= \exists \Delta.\texttt{N} \mid \exists \emptyset.\texttt{X} & \textit{existential types} \\
\texttt{B} & ::= \texttt{T} \mid \bot & \textit{type bounds} \\
\texttt{P} & ::= \texttt{T} \mid \star & \textit{method type parameter} \\
\\
\Delta & ::= \overline{\texttt{X} \rightarrow \texttt{[}\texttt{B}_L\texttt{-B}_U\texttt{]}} & \textit{type ranges} \\
\Gamma & ::= \overline{\texttt{x : T}} & \textit{var environments} \\
\texttt{X} & ::= \ldots & \textit{type vars} \\
\texttt{x} & ::= \ldots & \textit{expr vars} \\
\texttt{C} & ::= \ldots & \textit{class names}
\end{array}
$$

Lookup Functions:
Shared premise for lookup rules except F-OBJ:

$$CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{vX} \rightarrow \texttt{[}\texttt{B}_L\texttt{-B}_U\texttt{]}}\texttt{>} \lhd \texttt{N} \{\ \overline{\texttt{S f;}}\ \ \overline{M}\ \}$$

$$\mathit{fields}(\texttt{Object}) = \emptyset \qquad \text{(F-OBJ)}$$

$$\mathit{fields}(\texttt{C}) = \overline{\texttt{g}}, \overline{\texttt{f}}, \text{ if } \texttt{N} = \texttt{D<}\overline{\texttt{U}}\texttt{>} \text{ and } \mathit{fields}(\texttt{D}) = \overline{\texttt{g}} \qquad \text{(F-SUPER)}$$

$$\mathit{ftype}(\texttt{f}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{ftype}(\texttt{f}; \texttt{[}\overline{\texttt{T/X}}\texttt{]N}), \text{ if } \texttt{f} \notin \overline{\texttt{f}} \qquad \text{(FT-SUPER)}$$

$$\mathit{ftype}(\texttt{f}_i; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{[}\overline{\texttt{T/X}}\texttt{]S}_i, \qquad \text{(FT-CLASS)}$$

$$\mathit{mtype}(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m}; \texttt{[}\overline{\texttt{T/X}}\texttt{]N}), \text{ if } \texttt{m} \notin \overline{M} \qquad \text{(MT-SUPER)}$$

$$\mathit{mtype}(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{[}\overline{\texttt{T/X}}\texttt{]}(\texttt{<}\Delta\texttt{>}\ \overline{\texttt{U}} \rightarrow \texttt{U}), \quad \text{if } \texttt{<}\Delta\texttt{>}\ \texttt{U m(}\overline{\texttt{U x}}\texttt{)}\ \{\ \texttt{return e;}\ \} \in \overline{M} \qquad \text{(MT-CLASS)}$$

$$\mathit{mbody}(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mbody}(\texttt{m}; \texttt{[}\overline{\texttt{T/X}}\texttt{]N}), \text{ if } \texttt{m} \notin \overline{M} \qquad \text{(MB-SUPER)}$$

$$\mathit{mbody}(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \langle \overline{\texttt{x}}.\texttt{[}\overline{\texttt{T/X}}\texttt{]e} \rangle, \quad \text{if } \texttt{<}\Delta\texttt{>}\ \texttt{U m(}\overline{\texttt{U x}}\texttt{)}\ \{\ \texttt{return e;}\ \} \in \overline{M} \qquad \text{(MB-CLASS)}$$

Fig. 3. Syntax and Lookup Functions

4.1 Variance of a Type

Before we embark on the specifics of the VarJ formalism, we examine the essence of variance reasoning, i.e., how variances are computed in type expressions. For now, consider the subtyping relation of our formalism as a black box—it will be defined in Section 4.3. When is a type instantiation C<*Exp*1> a subtype of another instantiation C<*Exp*2>? We answer a more general question by defining a general predicate $var(\mathtt{X};\mathtt{T})$, where X is a type variable and T is a type expression. The goal of *var* is to determine: Given a type variable X and a type expression T that can contain X, what is the subtyping relationship between different "instantiations" of T with respect to (wrt) X, where an instantiation of T wrt to X is a substitution for X in T. For example, we want $var(\mathtt{X};\mathtt{T}) = +$ to imply $\mathtt{[U/X]T} <: \mathtt{[U'/X]T}$, if $\mathtt{U} <: \mathtt{U'}$.

To define *var*, we use predicate $\mathtt{v}(\mathtt{T};\mathtt{T'})$ as a notational shorthand, denoting the type of subtype relation between T and T′:

- $+(\mathtt{T};\mathtt{T'}) \equiv \mathtt{T} <: \mathtt{T'}$
- $-(\mathtt{T};\mathtt{T'}) \equiv \mathtt{T'} <: \mathtt{T}$
- $\mathbf{o}(\mathtt{T};\mathtt{T'}) \equiv +(\mathtt{T};\mathtt{T'}) \wedge -(\mathtt{T};\mathtt{T'})$
- $*(\mathtt{T};\mathtt{T'}) \equiv \mathit{true}$

Note that, by the variance lattice (in Figure 1), we have

$$\mathtt{v} \leq \mathtt{w} \implies \Big[\mathtt{v}(\mathtt{T};\mathtt{T'}) \implies \mathtt{w}(\mathtt{T};\mathtt{T'})\Big] \tag{1}$$

In general, we want for *var* the following property, which is a generalization of the subtype lifting lemma of Emir et al.'s modeling of definition-site variance [8]:

$$var(\mathtt{X};\mathtt{T}) = \mathtt{v} \implies \Big[\mathtt{v}(\mathtt{U};\mathtt{U'}) \implies \mathtt{[U/X]T} <: \mathtt{[U'/X]T}\Big] \tag{2}$$

By (1), (2) entails a more general implication:

$$\mathtt{v} \leq var(\mathtt{X};\mathtt{T}) \implies \Big[\mathtt{v}(\mathtt{U};\mathtt{U'}) \implies \mathtt{[U/X]T} <: \mathtt{[U'/X]T}\Big] \tag{3}$$

We assume there is a usual class table CT that maps class identifiers C to their definition (i.e. $CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{vX} \rightarrow \mathtt{[B}_L\mathtt{-B}_U\mathtt{]}}\mathtt{>} \lhd \mathtt{N}\ \{\quad \ldots\ \}$). Similarly, we define a variance table VT that maps class identifiers to their type parameters with their def-site variances. For example, assuming the class table mapping above, $VT(\mathtt{C}) = \overline{\mathtt{vX}}$. VT is overloaded to take an extra index parameter i to the i^{th} def-site variance annotation; e.g, if $VT(\mathtt{C}) = \overline{\mathtt{vX}}$, then $VT(\mathtt{C}, i) = \mathtt{v}_i$.

The expression $var(\mathtt{X};\mathtt{B})$ computes the variance of type variable X in type expression B. Figure 4 contains *var*'s definition. *var*'s type input is overloaded for non-existential types (R) and type ranges (Δ). ($var(\overline{\mathtt{X}};\phi)$ is further overloaded in the expected way for computing variances for sequences of type variables.)

The *var* relation is used in our type system to determine which variance is appropriate for each type expression. Eventually our proof connects it to the subtype relation, in Lemma 1. (Proofs of all key lemmas can be found in a technical report available at `http://people.cs.umass.edu/~jaltidor/ecoop12tr.pdf`.)

Variance of Types and Ranges: $var(\mathtt{X}; \phi)$, where $\phi ::= \mathtt{B} \mid \mathtt{R} \mid \Delta$

$var(\mathtt{X}; \mathtt{X}) = +$ (Var-XX)

$var(\mathtt{X}; \mathtt{Y}) = *$, if $\mathtt{X} \neq \mathtt{Y}$ (Var-XY)

$var(\mathtt{X}; \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \prod_{i=1}^{n} \big(\mathtt{v}_i \otimes var(\mathtt{X}; \mathtt{T}_i)\big)$, if $VT(\mathtt{C}) = \overline{\mathtt{vX}}$ (Var-N)

$var(\mathtt{X}; \bot) = *$ (Var-B)

$var(\mathtt{X}; \exists \Delta.\mathtt{R}) = var(\mathtt{X}; \Delta) \sqcap var(\mathtt{X}; \mathtt{R})$, if $\mathtt{X} \notin dom(\Delta)$ (Var-T)

$var(\mathtt{X}; \overline{\mathtt{Y} \to \mathtt{[B}_L\mathtt{-B}_U\mathtt{]}}) = \prod_{i=1}^{n} \Big(\big(- \otimes var(\mathtt{X}; \mathtt{B}_{Li})\big) \sqcap \big(+ \otimes var(\mathtt{X}; \mathtt{B}_{Ui})\big)\Big)$ (Var-R)

$\big[var(\overline{\mathtt{X}}; \phi) = \overline{\mathtt{v}}\big] \equiv \big[\forall i, var(\mathtt{X}_i; \phi) = \mathtt{v}_i\big]$, where $\phi ::= \mathtt{B} \mid \mathtt{R} \mid \Delta$ (Var-Seq)

Fig. 4. Variance of types and ranges

Lemma 1 (Subtype Lifting Lemma). If (a) $\overline{\mathtt{v}} \leq var(\overline{\mathtt{X}}; \mathtt{B})$ and (b) $\Delta \vdash \overline{\mathtt{v}(\mathtt{T}; \mathtt{U})}$ then $\mathtt{[}\overline{\mathtt{T}}\mathtt{/}\overline{\mathtt{X}}\mathtt{]B} <: \mathtt{[}\overline{\mathtt{U}}\mathtt{/}\overline{\mathtt{X}}\mathtt{]B}$.

We provide some intuition on the soundness of *var*'s definition. One "base case" of *var*'s definition is the Var-XX rule. To see why it returns $+$, note that the desired implication from the subtype lifting lemma holds for this case: if $+(\mathtt{T}; \mathtt{U})$, which is equivalent to $\mathtt{T} <: \mathtt{U}$, then $\mathtt{[T/X]X} = \mathtt{T} <: \mathtt{U} = \mathtt{[U/X]X}$. The Var-N rule computes the variance in a non-variable type using the $\otimes$ operator, which determines how variances compose, as described in Section 3. Var-R computes the variance of a type variable in a range. Computing the variance of ranges is necessary for computing the variance of constraints from type bounds on type parameters, which occur in existential types and method signatures. The domains of ranges are ignored by Var-R. A range becomes more "specialized" as the bounds get "squeezed". Informally, a range $\overline{[\mathtt{B}_L - \mathtt{B}_U]}$ is a *subrange* of range $\overline{[\mathtt{A}_L - \mathtt{A}_U]}$ if $\overline{\mathtt{A}_L <: \mathtt{B}_L}$ and $\overline{\mathtt{B}_U <: \mathtt{A}_U}$. The variance of the lower bound is transformed by contravariance to "reverse" the subtype relation, since we want the lower bound in the subrange to be a *supertype* of the lower bound in the superrange.[5] The subtype lifting lemma can be used to entail subrange relationships:

$$var(\mathtt{X}; \overline{\mathtt{Y} \to \mathtt{[B}_L\mathtt{-B}_U\mathtt{]}}) = \mathtt{v} \textbf{ and } \mathtt{v}(\mathtt{T}; \mathtt{U}) \implies \overline{\mathtt{[U/X]B}_L <: \mathtt{[T/X]B}_L} \textbf{ and } \overline{\mathtt{[T/X]B}_U <: \mathtt{[U/X]B}_U}$$

The variance of an existential type variable is just the meet of the variances of its range (Δ) and its body ($\mathtt{R}$). The Var-T rule has the premise "$\mathtt{X} \notin dom(\Delta)$" to follow *Barendregt's variable convention* [19], as in the TameFJ formalism. (*var* is undefined when this premise is not satisfied.) The variable convention substantially reduces the number of places requiring alpha-conversion to be applied and allows for more elegant proofs. The rules of the convention basically are: (1) relations in the type system are equivariant (respect alpha-renaming), and (2) no binder (declared variable) in a rule occurs free in the conclusion. For example, condition (1) holds for $var(\mathtt{X}; \mathtt{T})$ because we can rename binders

[5] Intuitively, the upper/lower bounds are in co-/contravariant positions, respectively.

to fresh names in existential types in T without changing the variance of X in T. Without the premise of rule VAR-T, this property would no longer hold. So that the important premises are more apparent, in the remaining rules we skip such "side-conditions" in the text and just mention that the premises for following the variable convention are implicit.

4.2 Variance of a Position

To see how *var* is used in our type system, we need to consider the variance of positions in a class definition. For example, return types are assumed to be in covariant positions while argument types are assumed to be in contravariant positions. These assumptions are used to type check class and method definitions and their def-site variance annotations.

The expressions "$\overline{\mathtt{v}} \leq \mathit{var}(\overline{\mathtt{X}}; \mathtt{B})$" and "$- \otimes \mathtt{v}$" are used frequently in the VarJ formalism. To connect our notation to previous work, we define the following:

$$\left[\overline{\mathtt{vX}} \vdash \mathtt{B}\ \mathit{mono}\right] \equiv \left[\overline{\mathtt{v}} \leq \mathit{var}(\overline{\mathtt{X}}; \mathtt{B})\right] \tag{4}$$

$$\left[\neg \mathtt{v}\right] = \left[- \otimes \mathtt{v}\right] \tag{5}$$

A "monotonicity" judgment of the syntactic form "$\overline{\mathtt{vX}} \vdash \mathtt{T}\ \mathit{mono}$" appears originally in Emir et al.'s definition-site variance treatment [8] and later in Kennedy and Pierce [13] as "$\overline{\mathtt{vX}} \vdash \mathtt{T}\ \mathit{ok}$". The semantics of these judgments in the aforementioned sources are similar to its definition here but differs in that they had no function similar to *var* nor a variance lattice. The negation operator $\neg$ also appears in [8] and [13] and is used to transform a variance by contravariance. Using the implications in Section 4.1, it is easy to show the following properties, which are important for type checking class definitions:

$$\mathtt{w} = \neg\mathtt{v} \implies \left[\mathtt{v}(\mathtt{B}, \mathtt{B}') \iff \mathtt{w}(\mathtt{B}', \mathtt{B})\right] \tag{6}$$

$$\overline{\mathtt{vX}} \vdash \mathtt{B}\ \mathit{mono} \implies \left[\overline{\mathtt{v}(\mathtt{T}, \mathtt{U})} \implies [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{B} <: [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{B}\right] \tag{7}$$

$$\overline{\neg\mathtt{vX}} \vdash \mathtt{B}\ \mathit{mono} \implies \left[\overline{\mathtt{v}(\mathtt{T}, \mathtt{U})} \implies [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{B} <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{B}\right] \tag{8}$$

Figure 5 contains rules for checking class and method definitions and the definition of the *override* predicate. Premises related to type checking with definition-site variance are highlighted. Auxiliarly lookup functions are used to compute the types of members (fields and methods) in class definitions. Their definitions are in Figure 3. These lookup functions take in non-variable types (N) instead of existential types. In the expression typing rules (in Figure 7), existential types are implicitly "unpacked" to non-variable types to type some expressions such as a field access. The process for packing and unpacking types is similar to the process performed in the TameFJ formalism. Section 4.4 has a brief overview of this process and an example type derivation.

Class and Method Typing:

$$\frac{\overline{\texttt{vX}} \vdash \texttt{N}, \overline{\texttt{T}}\ \mathit{mono} \qquad \Delta = \overline{\texttt{X} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]}} \qquad \emptyset \vdash \Delta\ \mathit{OK} \qquad \Delta \vdash \texttt{N}, \overline{\texttt{T}}\ \mathit{OK} \qquad \vdash \overline{M}\ \mathit{OK}\ \text{in}\ \texttt{C}}{\vdash \texttt{class C<}\overline{\texttt{vX} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]}}\texttt{>} \lhd \texttt{N}\ \{\ \overline{\texttt{T f;}}\ \ \overline{M}\ \}\ \mathit{OK}}\ \text{(W-Cls)}$$

$$\frac{\begin{array}{c}CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{vX} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]}}\texttt{>} \lhd \texttt{N}\ \{\ \ldots\ \} \qquad \Delta = \overline{\texttt{X} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]}} \\ \Delta \vdash \Delta'\ \mathit{OK} \qquad \Delta, \Delta' \vdash \texttt{T}, \overline{\texttt{T}}\ \mathit{OK} \\ \mathit{override}(\texttt{m}; \texttt{N}; \texttt{<}\Delta'\texttt{>}\ \overline{\texttt{T}} \to \texttt{T}) \qquad \overline{\neg\texttt{vX}} \vdash \overline{\texttt{T}}, \Delta'\ \mathit{mono} \qquad \overline{\texttt{vX}} \vdash \texttt{T}\ \mathit{mono} \\ \Delta, \Delta';\ \overline{\texttt{x} : \texttt{T}}, \texttt{this} : \exists\emptyset.\texttt{C<}\overline{\texttt{X}}\texttt{>} \vdash \texttt{e} : \texttt{T} \mid \emptyset\end{array}}{\vdash \texttt{<}\Delta'\texttt{>}\ \texttt{T m(}\overline{\texttt{T x}}\texttt{)}\ \{\ \texttt{return e;}\ \}\ \mathit{OK}\ \text{in}\ \texttt{C}}\ \text{(W-Meth)}$$

$$\frac{\mathit{mtype}(\texttt{m}; \texttt{N}) = \texttt{<}\Delta\texttt{>}\ \overline{\texttt{T}} \to \texttt{T}}{\mathit{override}(\texttt{m}; \texttt{N}; \texttt{<}\Delta\texttt{>}\ \overline{\texttt{T}} \to \texttt{T})}\ \text{(Over-Def)} \qquad \frac{\mathit{mtype}(\texttt{m}; \texttt{N})\ \text{is undefined}}{\mathit{override}(\texttt{m}; \texttt{N}; \texttt{<}\Delta\texttt{>}\ \overline{\texttt{T}} \to \texttt{T})}\ \text{(Over-Undef)}$$

Wellformed Ranges: $\Delta \vdash \Delta\ \mathit{OK}$

$$\frac{}{\Delta \vdash \emptyset\ \mathit{OK}}\ \text{(W-Rng-empty)} \qquad \frac{\texttt{X} \notin \mathit{dom}(\Delta) \qquad \Delta, \texttt{X} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]}, \Delta' \vdash \mathrm{B}_L, \mathrm{B}_U\ \mathit{OK} \qquad \Delta \vdash \mathit{ubound}_\Delta(\mathrm{B}_L) \sqsubset: \mathit{ubound}_\Delta(\mathrm{B}_U) \qquad \Delta \vdash \mathrm{B}_L <: \mathrm{B}_U \qquad \Delta, \texttt{X} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]} \vdash \Delta'\ \mathit{OK}}{\Delta \vdash \texttt{X} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]}, \Delta'\ \mathit{OK}}\ \text{(W-Rng)}$$

Non-Variable Upper Bound: $\mathit{ubound}_\Delta(\texttt{B})$

$$\mathit{ubound}_\Delta(\texttt{B}) = \begin{cases} \mathit{ubound}_\Delta(\mathrm{B}_U), \text{ if } \texttt{B} = \exists\emptyset.\texttt{X}, \text{ where } \Delta(\texttt{X}) = [\mathrm{B}_L - \mathrm{B}_U] \\ \texttt{B}, \text{ if } \texttt{B} = \exists\Delta'.\texttt{N} \end{cases}$$

Wellformed Types: $\Delta \vdash \phi\ \mathit{OK}$, where $\phi ::= \texttt{B} \mid \texttt{P} \mid \texttt{R}$

$$\frac{}{\Delta \vdash \texttt{Object<>}\ \mathit{OK}}\ \text{(W-Obj)} \qquad \frac{\texttt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \texttt{X}\ \mathit{OK}}\ \text{(W-X)} \qquad \frac{}{\Delta \vdash \bot\ \mathit{OK}}\ \text{(W-B)} \qquad \frac{}{\Delta \vdash \star\ \mathit{OK}}\ \text{(W-I)}$$

$$\frac{\texttt{class C<}\overline{\texttt{vX} \to \texttt{[}\mathrm{B}_L\texttt{-}\mathrm{B}_U\texttt{]}}\texttt{>} \lhd \texttt{N}\ \{\ \ \ldots\ \} \qquad \Delta \vdash \overline{\texttt{[T/X]}\mathrm{B}_L <: \texttt{T}} \qquad \Delta \vdash \overline{\texttt{T} <: \texttt{[T/X]}\mathrm{B}_U} \qquad \Delta \vdash \overline{\texttt{T}}\ \mathit{OK}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \mathit{OK}}\ \text{(W-N)} \qquad \frac{\Delta \vdash \Delta'\ \mathit{OK} \qquad \Delta, \Delta' \vdash \texttt{R}\ \mathit{OK}}{\Delta \vdash \exists\Delta'.\texttt{R}\ \mathit{OK}}\ \text{(W-T)}$$

Wellformed Expression Variable Environments $\Delta \vdash \Gamma\ \mathit{OK}$

$$\frac{}{\Delta \vdash \emptyset\ \mathit{OK}}\ \text{(W-Env-Empty)} \qquad \frac{\texttt{x} \notin \mathit{dom}(\Gamma) \qquad \Delta \vdash \texttt{T}\ \mathit{OK} \qquad \Delta \vdash \Gamma\ \mathit{OK}}{\Delta \vdash \Gamma, \texttt{x} : \texttt{T}\ \mathit{OK}}\ \text{(W-Env)}$$

Fig. 5. Wellformedness Judgments

The definition-site subtyping relation judgment $\Delta \vdash \texttt{N} \prec: \texttt{N}'$ is defined over non-variable types and considers definition-site annotations when concluding subtype relationships. For example, $VT(\texttt{C}) = +\texttt{X} \implies \Delta \vdash \texttt{C<}\exists\emptyset\texttt{.Dog>} \prec: \texttt{C<}\exists\emptyset\texttt{.Animal>}$, assuming $\Delta \vdash \exists\emptyset\texttt{.Dog} <: \exists\emptyset\texttt{.Animal}$. This relation is defined in Figure 6.

Definition-Site Subtyping: $\texttt{R} \prec: \texttt{R}$

$$\frac{\texttt{class C<}\overline{\texttt{vX} \to \texttt{[}B_L\texttt{-}B_U\texttt{]}}\texttt{>} \lhd \texttt{N}\ \{\ \ldots\ \} \quad \texttt{C} \neq \texttt{D} \quad \Delta \vdash \texttt{[}\overline{\texttt{T/X}}\texttt{]N} \prec: \texttt{D<}\overline{\texttt{U}}\texttt{>}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{D<}\overline{\texttt{U}}\texttt{>}}\ \text{(SD-Super)} \qquad \frac{VT(\texttt{C}) = \overline{\texttt{vX}} \quad \Delta \vdash \overline{\texttt{v(T, U)}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{C<}\overline{\texttt{U}}\texttt{>}}\ \text{(SD-Var)} \qquad \frac{}{\Delta \vdash \texttt{X} \prec: \texttt{X}}\ \text{(SD-X)}$$

Existential Subtyping: $\Delta \vdash \texttt{B} \sqsubset: \texttt{B}$

$$\frac{\Delta, \Delta' \vdash \texttt{N} \prec: \texttt{N}'}{\Delta \vdash \exists\Delta'.\texttt{N} \sqsubset: \exists\Delta'.\texttt{N}'}\ \text{(SE-SD)} \qquad \frac{}{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}}\ \text{(SE-Refl)} \qquad \frac{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}' \quad \Delta \vdash \texttt{B}' \sqsubset: \texttt{B}''}{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}''}\ \text{(SE-Tran)}$$

$$\frac{}{\Delta \vdash \bot \sqsubset: \texttt{B}}\ \text{(SE-Bot)} \qquad \frac{dom(\Delta') \cap fv(\exists\overline{\texttt{X} \to \texttt{[}B_L\texttt{-}B_U\texttt{]}}\texttt{.N}) = \emptyset \quad fv(\overline{\texttt{T}}) \subseteq dom(\Delta, \Delta') \quad \Delta, \Delta' \vdash \overline{\texttt{[T/X]}B_L} <: \texttt{T} \quad \Delta, \Delta' \vdash \texttt{T} <: \overline{\texttt{[T/X]}B_U}}{\Delta \vdash \exists\Delta'.\texttt{[}\overline{\texttt{T/X}}\texttt{]N} \sqsubset: \exists\overline{\texttt{X} \to \texttt{[}B_L\texttt{-}B_U\texttt{]}}\texttt{.N}}\ \text{(SE-Pack)}$$

Subtyping: $\Delta \vdash \texttt{B} <: \texttt{B}$

$$\frac{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}'}{\Delta \vdash \texttt{B} <: \texttt{B}'}\ \text{(ST-SE)} \qquad \frac{\Delta \vdash \texttt{B} <: \texttt{B}' \quad \Delta \vdash \texttt{B}' <: \texttt{B}''}{\Delta \vdash \texttt{B} <: \texttt{B}''}\ \text{(ST-Tran)} \qquad \frac{\Delta(\texttt{X}) = [B_L - B_U]}{\Delta \vdash B_L <: \exists\emptyset\texttt{.X}}\ \text{(ST-Lbound)} \quad \frac{}{\Delta \vdash \exists\emptyset\texttt{.X} <: B_U}\ \text{(ST-Ubound)}$$

Fig. 6. Subtyping Relations

The motivation for the assumed variances of positions is to ensure the subsumption principle holds for the subtyping hierarchy. Informally, if $\texttt{T} <: \texttt{U}$, then a value of type T may be provided whenever a value of type U is required. In the case of VarJ, the subsumption principle is established by showing appropriate subtype relationships between types of members from class definitions. Lemma 2 states a goal subsumption property, which is to have the type of field f of the supertype $\texttt{N}'$ become a more specific type for the subtype N. Although inherited fields syntactically have the same type as in the superclass definition, definition-site subtyping allows fields to have more specific types in the subtype. Lemma 3 states the goal subsumption property for types in method signatures; the sixth conclusion of this lemma holds because of the *override* predicate.

Lemma 2 (Subtyping Specializes Field Type).
If (a) $\vdash$ `class C<`$\overline{\texttt{vX} \to \texttt{[...]}}$`> ◁ N...` *OK* and (b) $\Delta \vdash$ `C<`$\overline{\texttt{T}}$`>` $\prec:$ `N′` and (c) *ftype*(`f`; `N′`) = `T`, then $\Delta \vdash$ *ftype*(`f`; `C<`$\overline{\texttt{T}}$`>`) $<:$ `T`.[6]

Lemma 3 (Subtyping Specializes Method Type).
If (a) $\vdash$ `class C<`$\overline{\texttt{vX} \to \texttt{[...]}}$`> ◁ N...` *OK* and (b) $\Delta \vdash$ `C<`$\overline{\texttt{T}}$`>` $\prec:$ `N′` and (c) *mtype*(`m`; `N′`) = `<`$\overline{\texttt{Y} \to \texttt{[}B_L\texttt{-}B_U\texttt{]}}$`>` $\overline{\texttt{U}} \to$ `U`, then: (1) *mtype*(`m`; `C<`$\overline{\texttt{T}}$`>`) = `<`$\overline{\texttt{Y} \to \texttt{[}A_L\texttt{-}A_U\texttt{]}}$`>` $\overline{\texttt{V}} \to$ `V`, (2) $\Delta \vdash$ `V` $<:$ `U`, (3) $\Delta \vdash \overline{\texttt{U} <: \texttt{V}}$, (4) $\Delta \vdash \overline{A_L <: B_L}$, (5) $\Delta \vdash \overline{B_U <: A_U}$, and (6) $var(\overline{\texttt{Y}}; \texttt{U}) = var(\overline{\texttt{Y}}; \texttt{V})$.

To satisfy the two lemmas above, we make assumptions about the variance of the positions that types can occur in. To *preserve* the subtype relationship order of a type in a member signature, we assume the type occurs in a *covariant* position (i.e., the subtype needs to have a more specific type appear in such a position). To *reverse* the subtype relationship order of a type in a member signature, we assume the type occurs in a *contravariant* position. The assumptions about the variance of the positions are reflected in the *mono* judgments in the W-Cls and W-Meth rules for checking class and method definitions. By (7), *not* negating the def-site variance annotations, $\overline{\texttt{v}}$, in the judgment "$\overline{\texttt{vX}} \vdash$ `T` *mono*" reflects that `T` is assumed to be in a covariant position. Since covariance, +, is the identity element for the $\otimes$ operator ($+ \otimes \texttt{v} = \texttt{v}$), the variances $\overline{\texttt{v}}$ do not need to be transformed by +. By (8), negating the def-site variance annotations in the judgment "$\overline{\neg\texttt{vX}} \vdash$ `T` *mono*" reflects that `T` is assumed to be in a contravariant position. We need to reverse the subtype relationship order for argument types and ranges in method type signatures. Negating the variance annotations for the argument types ensures the argument types are more general supertypes for the subtype.[7]

Negating the range of a method type signature ensures the range is *wider* for the subtype. For code examples motivating why ranges need to be widened for the subtype, see Section 2.4 of [8]. More generally, if (1) `e.<T>m()` type checks implying the type actual `T` is within the type bounds for `m`'s type argument and (2) *typeof*(`e′`) $<:$ *typeof*(`e`), then `e′.<T>m()` should type check as well even if `m` is overridden in the subclass. Hence, the subtype's version of `m` should accept a superset/wider range of types than accepted by the supertype's version of `m`.

4.3 Subtyping

Subtyping in VarJ is defined similarly to TameFJ. Figure 6 contains the subtyping rules. There are three levels of subtyping in VarJ, as in TameFJ. The first level of subtyping in TameFJ, the subclass relation, has been replaced with the definition-site subtyping relation $\prec:$ defined on non-existential types. Def-site

[6] If field assignments were allowed, then field types would be in both co- and contravariant positions, and both *ftype*(`f`; `C<`$\overline{\texttt{T}}$`>`) and *ftype*(`f`; `N′`) would be subtypes of each other.

[7] Bounds on class type parameters may make unrestricted use of type parameters by similar reasoning as in [8, p.7]. Once an object is created, they are forgotten.

subtyping is defined by the SD-* rules, which are similar to the subtyping rules from [13]. Like the subtype relation from [13], $\prec:$ is defined by syntax-directed rules[8] and shares the reflexive and transitive properties by similar reasoning as in [13]. The $\prec:$ judgment requires a typing context to check subtyping relationships between pairs of type actuals as done in the SD-Var rule.

The existential subtyping relation $\sqsubset:$ is defined by the SE-* rules and is similar to the "Extended subclasses" relation in TameFJ. The XS-Env rule from TameFJ was renamed to SE-Pack; it is the only subtyping rule that can pack (and actually also unpack) types into existential type variables. The XS-Sub-Class rule was not only renamed to SE-SD, but its premise was updated to use def-site subtyping. SE-SD allows def-site subtyping to be applied to both type variables in the type context Δ and existential type variables in Δ'. As a result, a type packed to an existential type variable may not be in the range of the variable. For example, if `Iterator` is covariant in its type parameter ($VT($`Iterator`$) = +$`X`), then the following subtype relationship is derivable: $\exists\emptyset$.`Iterator<PrettyDog>` $<:$ $\exists\emptyset$.`Iterator<Dog>` $<:$ $\exists$`Y` $\rightarrow$ `[Dog-Animal].Iterator<Y>`. Subtyping between two types implies the subsumption principle between the types. Since `Iterator<Dog>` can be packed to $\exists$`X` $\rightarrow$ `[Dog-Animal].Iterator<X>` and `Iterator<PrettyDog>` $<:$ `Iterator<Dog>`, it must be the case that `Iterator<PrettyDog>` can also be packed to $\exists$`X` $\rightarrow$ `[Dog-Animal].Iterator<X>`. This intuition is formalized in Lemma 4, which is similar to Lemma 35 from TameFJ, and establishes a relationship between existential subtyping and def-site subtyping.

Lemma 4 (Existential subtyping to def-site subtyping). If (a) $\Delta \vdash \exists\Delta'.$`R'` $\sqsubset:$ $\exists\overline{\texttt{X} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}.$`R` and (b) $\emptyset \vdash \Delta$ *OK*, then there exists $\overline{\texttt{T}}$ such that: (1) $\Delta, \Delta' \vdash$ `R'` $\prec:$ $\overline{\texttt{[T/X]R}}$ and (2) $\Delta, \Delta' \vdash \overline{\texttt{[T/X]B}_L} <: \overline{\texttt{T}}$ and (3) $\Delta, \Delta' \vdash \overline{\texttt{T}} <: \overline{\texttt{[T/X]B}_U}$ and (4) $fv(\overline{\texttt{T}}) \subseteq dom(\Delta, \Delta')$.

Existential subtyping does not conclude subtype relationships for type variables except for the reflexive case using SE-Refl. The (all) subtyping relation $<:$ allows non-reflexive subtype relationships with type variables by considering their bounds in the typing context. Since `T` or `U` may be type variables in a subtype relationship `T` $<:$ `U`, we want a stronger relationship between the non-variable upper bounds of `T` and `U`. Lemma 5 formalizes this notion and is similar to lemma 17 from TameFJ. The non-variable upper bound of a type `T` is $ubound_\Delta($`T`$)$, defined in Figure 5.

Lemma 5 (Subtyping to existential subtyping). If (a) $\Delta \vdash$ `T` $<:$ `T'` and (b) $\emptyset \vdash \Delta$ *OK* then $\Delta \vdash ubound_\Delta($`T`$) \sqsubset: ubound_\Delta($`T'`$)$.

4.4 Typing and Wildcard Capture

The expression typing rules in VarJ are mostly the same as in TameFJ and are given in Figure 7. Unlike TameFJ, VarJ allows method signatures to have lower

[8] The syntax-directed nature of these rules does not ensure that an algorithmic test of $\prec:$ is straightforward, because the premise of rule SD-Var appeals to the definition of the full $<:$ relation (hidden inside the `v` shorthand).

bounds. The *sift* function is needed for safe wildcard capture and is applied in the T-Invk rule for typing method invocations. The definition of *sift* required updating because of interaction with variant types. First, we give a brief overview of expression typing; see [4] for more thorough coverage.

Expression Typing. Consider the Java segment below. It type checks because the expression `box.elem` is typed as `String`. The type of `box.elem` is the same as the type actual passed to the `Box` type constructor. In this case, the type actual is "`? extends String`", which refers to some unknown subtype of `String`. To type `box.elem` with some known/named type, the most specific named type that can be assigned to `box.elem` is chosen, which is `String`.

```
class Box<E> { E elem; Box(E elem) { this.elem = elem; } }
Box<? extends String> box = ...
box.elem.charAt(0);
```

We explain this type derivation through the formal calculus. Types hidden by wildcards such as "`? extends String`" are "captured" as existential type variables. The type `Box<? extends String>` is modeled in VarJ by $\exists \mathtt{X} \rightarrow \mathtt{[}\bot\mathtt{\text{-}String]}.\mathtt{Box<X>}$. Expression typing judgments have the form $\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta'$. The second type variable environment Δ' is the *guard* of the judgment. It is used to keep track of type variables that have been unpacked from existential types during type checking. Variables in $dom(\Delta')$ may occur free in `T` and model hidden types. To type an expression without exposed (free) hidden types (existential type variables), the T-Subs rule is applied to find a suitable type without free existential type variables. The example typing derivation below illustrates this process on typing the "`box.elem`" expression from the previous code segment, where we assume $\Gamma = \mathtt{box} : \exists \mathtt{X} \rightarrow \mathtt{[}\bot\mathtt{\text{-}String]}.\mathtt{Box<X>}$.

$$\dfrac{\dfrac{\dfrac{\emptyset;\ \Gamma \vdash \mathtt{box} : \exists \mathtt{X} \rightarrow \mathtt{[}\bot\mathtt{\text{-}String]}.\mathtt{Box<X>} \mid \emptyset \qquad \mathit{ftype}(\mathtt{elem}; \mathtt{Box<X>}) = \mathtt{X}}{\emptyset;\ \Gamma \vdash \mathtt{box.elem} : \mathtt{X} \mid \mathtt{X} \rightarrow \mathtt{[}\bot\mathtt{\text{-}String]}}\ \text{(T-Field)} \qquad \dfrac{\emptyset, \mathtt{X} \rightarrow \mathtt{[}\bot\mathtt{\text{-}String]} \vdash \mathtt{X} <: \mathtt{String} \qquad \emptyset \vdash \mathtt{X} \rightarrow \mathtt{[}\bot\mathtt{\text{-}String]}\ \mathit{OK} \qquad \emptyset \vdash \mathtt{String}\ \mathit{OK}}{}}{\emptyset;\ \Gamma \vdash \mathtt{box.elem} : \mathtt{String} \mid \emptyset}\ \text{(T-Subs)}$$

Matching for Wildcard Capture. The T-Invk rule type checks a method invocation and uses *match* to perform wildcard capture. The definition of *match* is updated to use the definition-site subtyping relation ($\prec:$). Ignoring return types, consider a polymorphic method declared with type $\mathtt{<}\overline{\mathtt{Y}}\mathtt{>m(}\overline{\mathtt{U}}\mathtt{)}$ and called with types $\mathtt{<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\exists \Delta.\mathtt{R}}\mathtt{)}$. The parameters of $\mathit{match}(\overline{\mathtt{R}}; \overline{\mathtt{U}}; \overline{\mathtt{P}}; \overline{\mathtt{Y}}; \overline{\mathtt{T}})$ and their expected conditions are:

1. The bodies of the actual value argument types of a method invocation ($\overline{\mathtt{R}}$).
2. The formal argument value types of a method ($\overline{\mathtt{U}}$).
3. The *specified* type actuals of a method invocation ($\overline{\mathtt{P}}$).
4. The formal type arguments of a method ($\overline{\mathtt{Y}}$).
5. The *inferred* type actuals of a method invocation ($\overline{\mathtt{T}}$).

Expression Typing: $\Delta;\ \Gamma \vdash \texttt{e} \ :\ \texttt{T} \mid \Delta$

$$\frac{}{\Delta;\ \Gamma \vdash \texttt{x} \ :\ \Gamma(\texttt{x}) \mid \emptyset}\ \text{(T-Var)} \qquad \frac{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ OK \qquad \mathit{fields}(\texttt{C}) = \overline{\texttt{f}} \qquad \overline{\mathit{ftype}(f, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{U}} \qquad \Delta;\ \Gamma \vdash \texttt{e} \ :\ \texttt{U} \mid \emptyset}{\Delta;\ \Gamma \vdash \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \ :\ \exists\emptyset.\texttt{C<}\overline{\texttt{T}}\texttt{>} \mid \emptyset}\ \text{(T-New)}$$

$$\frac{\Delta;\ \Gamma \vdash \texttt{e} \ :\ \exists\Delta'.\texttt{N} \mid \emptyset \qquad \mathit{ftype}(\texttt{f};\texttt{N}) = \texttt{T}}{\Delta;\ \Gamma \vdash \texttt{e.f} \ :\ \texttt{T} \mid \Delta'}\ \text{(T-Field)} \qquad \frac{\Delta;\ \Gamma \vdash \texttt{e} \ :\ \texttt{U} \mid \Delta' \qquad \Delta, \Delta' \vdash \texttt{U} <: \texttt{T} \qquad \Delta \vdash \Delta'\ OK \qquad \Delta \vdash \texttt{T}\ OK}{\Delta;\ \Gamma \vdash \texttt{e} \ :\ \texttt{T} \mid \emptyset}\ \text{(T-Subs)}$$

$$\frac{\begin{array}{c}\Delta;\ \Gamma \vdash \texttt{e} \ :\ \exists\Delta'.\texttt{N} \mid \emptyset \qquad \mathit{mtype}(\texttt{m};\texttt{N}) = \texttt{<}\overline{\texttt{Y} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{>}\ \overline{\texttt{U}} \to \texttt{U} \\ \Delta \vdash \overline{\texttt{P}}\ OK \qquad \Delta;\ \Gamma \vdash \texttt{e} \ :\ \exists\Delta.\texttt{R} \mid \overline{\emptyset} \\ \mathit{sift}(\overline{\texttt{R}};\overline{\texttt{U}};\overline{\texttt{Y}}) = (\overline{\texttt{R}'};\overline{\texttt{U}'}) \qquad \mathit{match}(\overline{\texttt{R}'};\overline{\texttt{U}'};\overline{\texttt{P}};\overline{\texttt{Y}};\overline{\texttt{T}}) \\ \Delta'' = \Delta, \Delta', \overline{\Delta} \qquad \Delta'' \vdash \overline{\exists\emptyset.\texttt{R} <: \texttt{[}\overline{\texttt{T}/\texttt{Y}}\texttt{]U}} \\ \Delta'' \vdash \overline{\texttt{[}\overline{\texttt{T}/\texttt{Y}}\texttt{]B}_L <: \texttt{T}} \qquad \Delta'' \vdash \overline{\texttt{T} <: \texttt{[}\overline{\texttt{T}/\texttt{Y}}\texttt{]B}_U}\end{array}}{\Delta;\ \Gamma \vdash \texttt{e.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \ :\ \texttt{[}\overline{\texttt{T}/\texttt{Y}}\texttt{]U} \mid \Delta', \overline{\Delta}}\ \text{(T-Invk)}$$

Match:

$$\frac{\forall j,\ \texttt{P}_j = \star \implies \texttt{Y}_j \in \mathit{fv}(\overline{\texttt{R}'}) \qquad \forall i,\ \texttt{P}_i \neq \star \implies \texttt{T}_i = \texttt{P}_i \qquad \emptyset \vdash \overline{\texttt{R} \prec: \texttt{[}\overline{\texttt{T}/\texttt{Y}}\texttt{,}\ \overline{\texttt{T}'/\texttt{X}}\texttt{]R}'} \qquad \mathit{dom}(\overline{\Delta}) = \overline{\texttt{X}} \qquad \mathit{fv}(\overline{\texttt{T}}, \overline{\texttt{T}'}) \cap \overline{\texttt{Y}}, \overline{\texttt{X}} = \emptyset}{\mathit{match}(\overline{\texttt{R}}; \overline{\exists\Delta.\texttt{R}'}; \overline{\texttt{P}}; \overline{\texttt{Y}}; \overline{\texttt{T}})}\ \text{(Match)}$$

Sift: $\mathit{sift}(\overline{\texttt{R}};\overline{\texttt{U}};\overline{\texttt{Y}}) = (\overline{\texttt{R}'};\overline{\texttt{U}'})$

$$\frac{}{\mathit{sift}(\emptyset;\emptyset;\overline{\texttt{Y}}) = (\emptyset;\emptyset)}\ \text{(Sift-Empty)} \qquad \frac{\overline{\texttt{Y}} \cap \mathit{fv}(\texttt{U}) = \overline{\texttt{X}} \qquad \mathit{var}(\overline{\texttt{X}};\texttt{U}) = \overline{o} \qquad \mathit{sift}(\overline{\texttt{R}};\overline{\texttt{U}};\overline{\texttt{Y}}) = (\overline{\texttt{R}'};\overline{\texttt{U}'})}{\mathit{sift}((\texttt{R},\overline{\texttt{R}});(\texttt{U},\overline{\texttt{U}});\overline{\texttt{Y}}) = ((\texttt{R},\overline{\texttt{R}'});(\texttt{U},\overline{\texttt{U}'}))}\ \text{(Sift-Add)}$$

$$\frac{\overline{\texttt{Y}} \cap \mathit{fv}(\texttt{U}) = \overline{\texttt{X}} \qquad \mathit{var}(\texttt{X}_j;\texttt{U}) \neq o,\ \text{for some } \texttt{X}_j \in \overline{\texttt{X}} \qquad \mathit{sift}(\overline{\texttt{R}};\overline{\texttt{U}};\overline{\texttt{Y}}) = (\overline{\texttt{R}'};\overline{\texttt{U}'})}{\mathit{sift}((\texttt{R},\overline{\texttt{R}});(\texttt{U},\overline{\texttt{U}});\overline{\texttt{Y}}) = (\overline{\texttt{R}'};\overline{\texttt{U}'})}\ \text{(Sift-Skip)}$$

Fig. 7. Expression Typing and Auxiliary Functions For Wildcard Capture

Figure 8 contains the reduction rules for performing runtime evaluation. The R-Invk rule also uses *match* to compute inferred type actuals because some of the specified type actuals ($\overline{\mathtt{P}}$) may be the type inference marker $\star$. Since each occurrence of the $\star$ marker may refer to different types, *match* is needed to compute the concrete types to substitute for the formal type arguments' ($\overline{\mathtt{Y}}$) occurrences in the method body.

Computation Rules: $\mathtt{e} \mapsto \mathtt{e}$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}}{\mathtt{new\ C\langle\overline{T}\rangle(\overline{v}).f}_i \mapsto \mathtt{v}_i} \quad \text{(R-Field)}$$

$$\frac{\begin{array}{c}\mathtt{v} = \mathtt{new\ N(\overline{v'})} \qquad \overline{\mathtt{v} = \mathtt{new\ N(\overline{v''})}} \qquad \mathit{mbody}(\mathtt{m};\mathtt{N}) = \langle\overline{\mathtt{x}}.\mathtt{e}_0\rangle \\ \mathit{mtype}(\mathtt{m};\mathtt{N}) = \mathtt{\langle \overline{Y} \to [B_L\text{-}B_U]\rangle\ \overline{U} \to U} \\ \mathit{sift}(\overline{\mathtt{N}};\overline{\mathtt{U}};\overline{\mathtt{Y}}) = (\overline{\mathtt{N'}};\overline{\mathtt{U'}}) \qquad \mathit{match}(\overline{\mathtt{N'}};\overline{\mathtt{U'}};\overline{\mathtt{P}};\overline{\mathtt{Y}};\overline{\mathtt{T}})\end{array}}{\mathtt{v.\langle\overline{P}\rangle m(\overline{v}) \mapsto [\overline{v/x}, v/this, \overline{T/Y}]e_0}} \quad \text{(R-Invk)}$$

Congruence Rules: $\mathtt{e} \mapsto \mathtt{e}$

$$\frac{\mathtt{e} \mapsto \mathtt{e'}}{\mathtt{e.f} \mapsto \mathtt{e'.f}} \quad \text{(RC-Field)} \qquad \frac{\mathtt{e}_i \mapsto \mathtt{e}'_i}{\mathtt{new\ C\langle\overline{T}\rangle(..e}_i\mathtt{..)} \mapsto \mathtt{new\ C\langle\overline{T}\rangle(..e}'_i\mathtt{..)} \quad \text{(RC-New-Arg)}$$

$$\frac{\mathtt{e} \mapsto \mathtt{e'}}{\mathtt{e.\langle\overline{P}\rangle m(\overline{e}) \mapsto e'.\langle\overline{P}\rangle m(\overline{e})}} \quad \text{(RC-Inv-Recv)} \qquad \frac{\mathtt{e}_i \mapsto \mathtt{e}'_i}{\mathtt{e.\langle\overline{P}\rangle m(..e}_i\mathtt{..) \mapsto e.\langle\overline{P}\rangle m(..e}'_i\mathtt{..)} \quad \text{(RC-Inv-Arg)}$$

Fig. 8. Reduction Rules

Sifting for Wildcard Capture. The *sift* function is used in VarJ and TameFJ to filter inputs passed to *match* (in the T-Invk and R-Invk rules). The goal of *sift* is to only allow inference from types that are in "fixed" or invariant positions. Without applying *sift*, counter examples to the subject reduction (type preservation) theorem can result. First, note that the following two judgments are derivable.

1. $\mathit{match}(\mathtt{Dog}; \exists\emptyset.\mathtt{Y}; \star; \mathtt{Y}; \mathtt{Dog})$ (mainly) because $\mathtt{Dog} \prec: \mathtt{[Dog/Y]Y} = \mathtt{Dog}$.
2. $\mathit{match}(\mathtt{Dog}; \exists\emptyset.\mathtt{Y}; \star; \mathtt{Y}; \mathtt{Animal})$ (mainly) because $\mathtt{Dog} \prec: \mathtt{[Animal/Y]Y} = \mathtt{Animal}$.

Assume `List` is invariant and consider the following.

```
<X> List<X> createList(X arg) { return new List<X>(); }
```

```
createList<*>(new Dog()) : List<Animal>
↦ new List<Dog>() : List<Dog>
```

The expression `createList<*>(new Dog())` can be typed with `List<Animal>` because `new Dog() : Animal`, and the inferred type actual used for typing the expression can be `Animal`. However, the inferred type used for typing the method invocation is not required to be the same inferred type, computed in the R-Invk rule, substituted into the method body. Without *sift*, the above evaluation step is possible, which contradicts the subject reduction theorem, since, by the invariance of `List`, `new List<Dog>()` cannot be typed with `List<Animal>`.

In TameFJ, *sift* filters out a pair of a type actual body `R` and a formal type `U`, if $\mathtt{U} = \exists\emptyset.\mathtt{X}$ and `X` is one of the formal type arguments ($\overline{\mathtt{Y}}$). Due to *sift*, the two *match* judgments above could never be derived in TameFJ. Moreover, TameFJ allows an existential type variable to be passed as parameter for a formal type variable argument only if the formal type variable is used as a type parameter. Since every type constructor in TameFJ is assumed to be invariant, every type variable used for inference is in an invariant position. This no longer holds in VarJ with variant type constructors. If we assume `Iterator` is covariant, a counter example similar to the previous one can be produced with the following method:

```
<X> List<X> createList2(Iterator<X> arg) { return new List<X>(); }
```

Hence, we update the definition of *sift* to use *var* to check if a method type parameter occurs at most invariantly. We find the restriction of not allowing wildcard capture in variant positions not to be practically restrictive. A wildcard type for a variant type typically has an equivalent non-wildcard type. `Iterator<?>` is equivalent to `Iterator<Object>` by covariance of `Iterator`. `BiGeneric<?>` is equivalent to `BiGeneric<T>`, for any `T`, if `BiGeneric` is bivariant. In such cases, the need for wildcard capture is eliminated because the required type actuals to specify in a method call can be named. The VarJ grammar does not allow the bottom type $\bot$ to be specified as a type actual. However, we have not found any practical need for wildcard capture with contravariant types.

4.5 Type Soundness

We prove type soundness for VarJ by proving the progress and subject reduction theorems below. As in TameFJ, a non-empty guard is required in the statement of the progress theorem when applying the inductive hypothesis in the proof for the case when the T-Subs rule is applied.

Theorem 1 (Progress). For any $\Delta, \mathtt{e}, \mathtt{T}$, if $\emptyset;\ \emptyset \vdash \mathtt{e} : \mathtt{T} \mid \Delta$, then either $\mathtt{e} \mapsto \mathtt{e}'$ or there exists a `v` such that $\mathtt{e} = \mathtt{v}$.

Theorem 2 (Subject Reduction). For any $\mathtt{e}, \mathtt{e}', \mathtt{T}$, if $\emptyset;\ \emptyset \vdash \mathtt{e} : \mathtt{T} \mid \emptyset$ and $\mathtt{e} \mapsto \mathtt{e}'$, then $\emptyset;\ \emptyset \vdash \mathtt{e}' : \mathtt{T} \mid \emptyset$.

The key difficulty in proving these theorems can be captured by a small number of key lemmas whose proofs are substantially affected by variance reasoning. Lemma 6 is probably the main one, which relates subtyping and wildcard capture, and is similar to lemma 36 from [4]. It states that the method receiver's ability to perform wildcard capture is preserved in subtypes with respect to the

method receiver. (A similar lemma holds for method arguments.) It shows that the subsumption principle holds even under interaction with wildcard capture.

Lemma 6 (Subtyping Preserves *match*ing (receiver)). If (a) $\Delta \vdash \exists\Delta_1.\mathtt{N}_1 \sqsubset: \exists\Delta_2.\mathtt{N}_2$ and (b) $mtype(\mathtt{m};\mathtt{N}_2) = \mathtt{<}\overline{\mathtt{Y}_2 \to \mathtt{[B}_{2L}\mathtt{-B}_{2U}\mathtt{]}}\mathtt{>}\ \overline{\mathtt{U}_2} \to \mathtt{U}_2$ and (c) $mtype(\mathtt{m};\mathtt{N}_1) = \mathtt{<}\overline{\mathtt{Y}_1 \to \mathtt{[B}_{1L}\mathtt{-B}_{1U}\mathtt{]}}\mathtt{>}\ \overline{\mathtt{U}_1} \to \mathtt{U}_1$ and (d) $sift(\overline{\mathtt{R}};\overline{\mathtt{U}_2};\overline{\mathtt{Y}_2}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'_2})$ and (e) $match(\overline{\mathtt{R}'};\overline{\mathtt{U}'_2};\overline{\mathtt{P}};\overline{\mathtt{Y}_2};\overline{\mathtt{T}})$ and (f) $\emptyset \vdash \Delta\ OK$ and (g) $\Delta, \Delta' \vdash \overline{\mathtt{T}}\ OK$ then: (1) $sift(\overline{\mathtt{R}};\overline{\mathtt{U}_1};\overline{\mathtt{Y}_1}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'_1})$ and (2) $match(\overline{\mathtt{R}'};\overline{\mathtt{U}'_1};\overline{\mathtt{P}};\overline{\mathtt{Y}_1};\overline{\mathtt{T}})$.

5 Discussion

Boundary Analysis. Definition-site variance can imply that the variance of a type does not depend on all of the type bounds that occur in the type. Our earlier work [1] presented a definition of $var(\mathtt{X};\mathtt{U})$ that performed a simple *boundary analysis* to compute such irrelevant bounds. As discussed in Section 3.1, if generic `C<Y>` is covariant wrt to `Y`, then the lower bound of a use-site variant instantiation is ignored, which is sound for the VarLang calculus [1]: $var(\mathtt{X};\mathtt{C<-T>}) = (+ \sqcup -) \otimes var(\mathtt{X};\mathtt{T}) = * \otimes var(\mathtt{X};\mathtt{T}) = *$. Hence, $var(\mathtt{X};\mathtt{C<-T>}) = *$, even if `X` occurred in the lower bound, `T`.

The ability to ignore type bounds is present in a disciplined way in our VarJ formalism, although there is no explicit variance joining mechanism in the definition of *var*. For example, if `Iterator` is covariant in its type parameter, we can infer the following (where the notation $\mathtt{T} \equiv \mathtt{U}$ denotes $\mathtt{T} <: \mathtt{U} \wedge \mathtt{U} <: \mathtt{T}$): $\exists\mathtt{X} \to \mathtt{[Dog\text{-}Animal].Iterator<X>} \equiv \exists\mathtt{X} \to \mathtt{[}\bot\mathtt{\text{-}Animal].Iterator<X>}$. Clearly, $\exists\mathtt{X} \to \mathtt{[Dog\text{-}Animal].Iterator<X>} <: \exists\mathtt{X} \to \mathtt{[}\bot\mathtt{\text{-}Animal].Iterator<X>}$ because the range of the type variable is wider in the supertype. The inverse is derivable by applying a combination of the SE-SD, SE-Pack, and ST-* rules:

$$\exists\mathtt{X} \to \mathtt{[}\bot\mathtt{\text{-}Animal].Iterator<X>} <: \exists\mathtt{X} \to \mathtt{[}\bot\mathtt{\text{-}Animal].Iterator<Animal>}$$
$$<: \exists\emptyset.\mathtt{Iterator<Animal>} <: \exists\mathtt{X} \to \mathtt{[Dog\text{-}Animal].Iterator<X>}$$

As we saw in Section 3.2, this reasoning is not sound in the presence of F-bounded polymorphism. It is important to realize that the issue with recursive bounds is not specific to the use of bounds in type definitions.[9] The counterexample of Section 3.2 used `interface Trouble<P extends List<P>> extends Iterator<P> {}`. However, even if we restrict our attention to a plain `Iterator` (or, equivalently, if the class type bound, `extends List<P>`, of `Trouble` is removed) it is still *not* safe to assume the following subtype relation, by reasoning similar to that used in Section 3.2:
$\exists\mathtt{X} \to \mathtt{[YourList\text{-}List<X>].Iterator<X>} <: \exists\mathtt{X} \to \mathtt{[MyList\text{-}List<X>].Iterator<X>}$.
The above subtype relationship would violate the subsumption principle. The

[9] In our earlier work [1], when we performed an application to Java it sufficed to be overly conservative at this point: the mere appearance of a type variable in the upper bound of a type definition caused us to consider the definition as invariant relative to this type variable. For VarJ, which is richer in terms of where bounds can appear, even this kind of conservatism is not sufficient.

latter type can return a $\exists$`X` $\rightarrow$ `[MyList-List<X>].List<X>` from its `next` method, but the former type cannot because
$\exists$`X` $\rightarrow$ `[YourList-List<X>].List<X>` $\not<:$ $\exists$`X` $\rightarrow$ `[MyList-List<X>].List<X>`, by the invariance of `List`. In contrast to the earlier, correct subtyping, VarJ does not support the above erroneous subtyping because it cannot establish that the upper bounds of the two instatiations of `Iterator` are related: we cannot derive that $\exists$`X` $\rightarrow$ `[YourList-List<X>].List<X>` is a subtype of some non-existential type, $\exists\emptyset$`.List<T>`, which is, in turn, a subtype of $\exists$`X` $\rightarrow$ `[MyList-List<X>].List<X>`.

Contrasting the two examples shows that boundary analysis is complex and can be unintuitive to the programmer. Note, however, that the VarJ calculus merely tells us what is possible to infer correctly. A practical implementation may choose not to perform all possible inferences. A specific scenario is that of separating boundary analysis from type checking. Useless bounds can be "removed" during a preprocessing step performed *before* type checking. This is analogous to general type inference algorithms relative to type checking algorithms: type checking can be performed independently of the type inference performed to compute type annotations skipped by programmers. Our variance-based type checking can be performed independently of the "useless boundary analysis". For example, a boundary preprocessing step could transform input type $\exists$`X` $\rightarrow$ `[Dog-Animal].Iterator<X>` to the equivalent type $\exists$`X` $\rightarrow$ `[`$\bot$`-Animal].Iterator<X>`. This opens the door to many practical instantiations—e.g., an optimistic but possibly unsound bound inference inside an IDE (which interacts with the user, offering immediate feedback and suggesting relaxations of expressions the user types in) combined with a simple but sound checking inside the compiler.

Definition-Site Variance and Erasure. A practical issue with definition-site variance concerns its use with an erasure-based translation. Consider a covariant `class A<+X>` {...} and an invariant subtype `class B<oX> extends A<X>` {...}. We can then have:

```
A<Integer> a = new B<Integer>;
A<Object> a2 = a;  // fine by covariance of A
B<Object> b = (B<Object>) a2;
```

In a language with an expansion-based translation, such as C#, the last line will fail dynamically: an object with dynamic type `B<Integer>` cannot be cast to a `B<Object>`. In an erasure-based translation, however, the cast cannot check the type parameter (which has been erased) and will therefore succeed, causing errors further down the road. (In this case, a runtime error could result from a non-cast expression, thus violating type soundness.) This practical consideration affects all type systems that combine variance, casts, and erasure. For instance, Scala already handles such cases with a static type warning. Effectively, no cast to a subtype with tighter variance is safe. The result is somewhat counter-intuitive in that it defies common patterns for safe casting. For instance, the cast could have been performed after an "`a2 instanceof B<Object>`" check to establish that `a2` is indeed of type `B<Object>`. In this case the programmer would think that the cast warning can be ignored, which is not the case. In practice, any deployment

of our type system in an erasure-based setting would have to follow the same policy as Scala regarding cast warnings.

6 Related Work

Definition-site variance was first investigated in the late 80's [7,2,3] when parametric types were incorporated into object-oriented languages. It has recently experienced a resurgence as newer languages such as Scala [14] and C# [9] chose it as means to support variant subtyping. Perhaps surprisingly, with such a long history, it has only recently been formalized and proven sound in a non-toy setting [8].

Use-site variance was introduced by Thorup and Torgersen [17] in response to the rigidity in class definitions imposed by definition-site variance. The concept was later generalized and formalized by Igarashi and Viroli [10]. The elegance and flexibility of the approach evoked a great deal of enthusiasm, and was quickly introduced into Java [18]. The same flexibility also proved challenging to both researchers and practitioners. The soundness of wildcards in Java has only recently been proven [4], and the implementation of wildcards has been mired in issues [5,15,16].

The work of Viroli and Rimassa [20] attempts to clarify when variance is to be used, introducing concepts of produce/consume, which are an improvement over the write/read view. Our approach offers a generalization and a high-level way to reason soundly about the variance of a type. Other recent work discusses the complex relationship between type-erasure and wildcards [6], as well as the concept of variance at the level of tuning access to a path type in tree-like class definitions [12].

7 Conclusion

This paper presented VarJ, the first formal model for Java with definition-site variance, wildcards, and intricate features such as wildcard capture. VarJ gives a framework for reasoning about the variance of various types (e.g., bounded existential types). We presented theory underlying the assumed variances of the positions that types can occur in (e.g., the upper bound of a method type parameter is contravariant). Thus, our calculus resolves questions that are central in the design of any language involving parametric polymorphism and subtyping.

Acknowledgments. We thank the anonymous ECOOP reviewers for their feedback, Ross Tate for providing examples on the complications of wildcards, Nicholas Cameron for discussions on TameFJ and on performing type inference at runtime, Andrew Kennedy for discussions about the C# formalism with definition-site variance, and Christian Urban for clarifying Barendregt's variable convention. This work was funded by the National Science Foundation under grants CCF-0917774 and CCF-0934631.

References

1. Altidor, J., Huang, S.S., Smaragdakis, Y.: Taming the wildcards: Combining definition- and use-site variance. In: Programming Language Design and Implementation, PLDI (2011)

2. America, P., van der Linden, F.: A parallel object-oriented language with inheritance and subtyping. In: European Conf. on Object-Oriented Programming and Object-Oriented Programming Systems, Languages, and Applications, OOPSLA/ECOOP (1990)
3. Bracha, G., Griswold, D.: Strongtalk: typechecking smalltalk in a production environment. In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (1993)
4. Cameron, N., Drossopoulou, S., Ernst, E.: A Model for Java with Wildcards. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 2–26. Springer, Heidelberg (2008)
5. Chin, W.-N., Craciun, F., Khoo, S.-C., Popeea, C.: A flow-based approach for variant parametric types. In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2006)
6. Cimadamore, M., Viroli, M.: Reifying wildcards in Java using the EGO approach. In: SAC 2007: Proceedings of the 2007 ACM Symposium on Applied Computing (2007)
7. Cook, W.: A proposal for making Eiffel type-safe. In: European Conf. on Object-Oriented Programming, ECOOP (1989)
8. Emir, B., Kennedy, A., Russo, C.V., Yu, D.: Variance and Generalized Constraints for C$^{\#}$ Generics. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 279–303. Springer, Heidelberg (2006)
9. Hejlsberg, A., Wiltamuth, S., Golde, P.: C# Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
10. Igarashi, A., Viroli, M.: On Variance-Based Subtyping for Parametric Types. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 441–469. Springer, Heidelberg (2002)
11. Igarashi, A., Viroli, M.: Variant parametric types: A flexible subtyping scheme for generics. ACM Trans. Program. Lang. Syst. 28(5), 795–847 (2006)
12. Igarashi, A., Viroli, M.: Variant path types for scalable extensibility. In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2007)
13. Kennedy, A.J., Pierce, B.C.: On decidability of nominal subtyping with variance, 2006. In: FOOL-WOOD 2007 (2007)
14. Odersky, M.: The Scala Language Specification v 2.8 (2010)
15. Smith, D., Cartwright, R.: Java type inference is broken: can we fix it? In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2008)
16. Tate, R., Leung, A., Lerner, S.: Taming wildcards in Java's type system. In: Programming Language Design and Implementation, PLDI (2011)
17. Thorup, K.K., Torgersen, M.: Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 186–204. Springer, Heidelberg (1999)
18. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahe, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: SAC 2004: Proc. of the 2004 Symposium on Applied Computing (2004)
19. Urban, C., Berghofer, S., Norrish, M.: Barendregt's Variable Convention in Rule Inductions. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 35–50. Springer, Heidelberg (2007)
20. Viroli, M., Rimassa, G.: On access restriction with Java wildcards. Journal of Object Technology 4(10), 117–139 (2005)

Constraint-Based Refactoring with Foresight

Friedrich Steimann and Jens von Pilgrim

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org, Jens.vonPilgrim@feu.de

Abstract. Constraint-based refactoring tools as currently implemented generate their required constraint sets from the programs to be refactored, before any changes are performed. Constraint generation is thus unable to see — and regard — the changed structure of the refactored program, although this new structure may give rise to new constraints that need to be satisfied for the program to maintain its original behaviour. To address this problem, we present a framework allowing the constraint-generation process to foresee all changes a refactoring might perform, generating — at the outset of the refactoring — all constraints necessary to constrain these changes. As we are able to demonstrate, the computational overhead imposed by our framework, although threatening viability in theory, can be reduced to tractable sizes.

1 Introduction

Refactoring is the discipline of changing a program in such a way that one or more of its non-functional properties (readability, maintainability, etc.) are improved, while its behaviour is maintained [4]. When applied to real programs written in real programming languages, refactoring involves complex precondition checking and mechanics that contain deeply nested case analyses, making refactoring without tool support tedious and error-prone. A steadily growing number of fully automated refactoring tools is therefore being devised; of these, a considerable part is constraint based (e.g., [1, 3, 5, 9, 10, 16–20]).

Current approaches to constraint-based refactoring use so-called *constraint rules* to generate sets of constraints from the programs to be refactored. The generated constraints rule over how the program may be changed without affecting its well-formedness or behaviour. Constraints are said to be generated in a syntax-directed manner [1, 5, 10, 20], i.e., based on the program's abstract syntax tree (AST), which represents the program as is before the refactoring.

One problem of this approach is that constraint rule application is unable to see the structural changes of the AST imposed by a refactoring. For instance, if a refactoring moves a program element to another location (corresponding to another node in the AST), the constraints imposed on the element by this new location have not been generated, since at the time of rule application, the element was still in its old location. Note that taking the *refactoring intent* (here to move the element to a known location) into account does not generally suffice to fix the problem, since the intended

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 535–559, 2012.

refactoring may require other elements to change as well (here: to move to the same location), which precisely only being known *after* having solved the constraints describing the refactoring problem. Thus, constraint-based refactorings need some kind of recognition of how a program is going to change.

In this paper, we present a solution to a class of problems that, following our first mention of it in [17], we have dubbed the *foresight problem* of constraint-based refactoring. Our solution relies on constraint rule rewriting and quantified constraints extending the scope and expressiveness of constraint rules so that they can cover all possible changes of a program's structure that might infringe its well-formedness or affect its behaviour. To address the computational complexity introduced by this, we present an algorithm for cancelling constraints not needed for a concrete refactoring. That our approach is indeed viable is demonstrated by applying it to five different variants of the PULL UP FIELD refactoring.

The remainder of this paper is organized as follows. In Section 2, we motivate our work by presenting various examples of the foresight problem, and reflect on the related literature. In Section 3 we provide a quick introduction to constraint-based refactoring, mostly for readers not acquainted with this refactoring technique. Sections 4 and 5 introduce our notions of quantified constraints and constraint rule rewriting that will be exploited in Section 6 for addressing the foresight problem (including the complexity imposed by our solution). After a brief sketch of the implementation in Section 7, our evaluation in Section 8 shows that the added complexity can be reduced significantly for many practical cases.

2 Motivation

To give the reader an impression of how constraint-based refactoring works, we first take a look at the following simple

EXAMPLE 1: Consider the Java program

```
class A {}
class B extends A {
  int i = 1;
  int j = this.i;
}
```

and the intended refactoring "pull up field `j` from class `B` to class `A`". A quick analysis shows that this is not possible if the pulling up of `j` is the only change the refactoring is allowed to make; if it may perform additional changes, it may be made to work by pulling up field `i` as well, or by separating the declaration of `j` from its initialization. An indeed, Eclipse (whose implementation of the PULL UP FIELD refactoring is constraint-based [19]) warns the user of the fact that field `i` will be undefined in the new location of `j`. ♦

In constraint-based refactoring, what a refactoring tool must do in order to perform the refactoring correctly is described as a *constraint satisfaction problem* (CSP) generated from the program as is, and the refactoring intent. The two constraints sufficiently describing the refactoring problem of the above example are that

1. the declaring type of `j`, the location (i.e., the hosting type) of `this`, and the location of the reference to `i` must be equal (since they are part of the same statement); and

2. the location of this must be a (non-strict) subtype of the declaring type of i (so that i is defined, either directly or via inheritance, for the object represented by this).

Note that both constraints are satisfied by the program as is:

1. the declaring type of j, the location of this, and the location of the reference to i are all B and
2. the location of this, B, is a (non-strict) subtype of the declaring type of i, also B.

However, the constraints are not satisfied after the declaration of j has been pulled up to A, since then the declaring type of j becomes A which, by satisfaction of Constraint 1, implies that the location of this is also A, which violates Constraint 2, since A is not a subtype of the declaring type of i, B. Both constraints can be satisfied, however, by changing the declaring type of i to A also, which is equivalent to pulling up i as well.

In current implementations of constraint-based refactoring, constraints like the above are generated from a program to be refactored by application of so-called *constraint rules*, whose precedents are matched against the AST representation of the program as is before the refactoring. As the following examples will demonstrate, this approach (which worked fine for Example 1) is challenged by refactorings that change the structure of the AST, by changing the locations of program elements.

2.1 Examples of the Foresight Problem

We begin our exploration of the foresight problem with the following simple

EXAMPLE 2: In the sample program

```
package p;
public class A {}
package q;
class B extends p.A { protected int i; }
class C extends B { void m(B b) { i = b.i; } }
```

pulling up i from B to A seems possible at first glance (since generally, *protected* accessibility suffices for inheritance across different packages), but will cause a compile error on b.i in C.m(B), since a rule of the Java language specification (JLS; [6], §6.6.2) mandates that

if a member (here: A.i) of an object (b) is accessed (b.i) from outside (q) the package in which it is declared (p) by code (C.m(B)) that is not responsible for the implementation of that object,
then accessibility must be *public*.

However, without special measures this rule (which corresponds to the rule Acc-2 of [17]) fails to generate the constraint required for adjusting i's accessibility to *public*. ♦

The problem exposed by Example 2 is that with the program as is, one conjunct of the rule precedent, that the access occurs from outside the package, is not fulfilled (since the declaration of i and its access in C.m are in the same package when the rule is applied), so that no constraint requiring *public* accessibility will be generated by applying this rule to the program. That *public* accessibility is required for i is known only after the fact, namely after it has been pulled up. Generation of the corresponding constraint thus requires foresight of the move.

Of course, one could argue that required accessibility always depends on the location of the accessor and the accessed, and that any constraint constraining accessibility should take the variability of the locations of the two into account. This is somewhat different for the following

EXAMPLE 3: In the sample program

```
class A {}
class B extends A { static int i = 0; }
interface I { int i = 1; }
class C extends A implements I {
  static int j = i;
}
```

pulling up the field `i` from class `B` to class `A` makes the access of `i` from class `C` ambiguous, since it is unclear whether `I.i` or `A.i` is referenced. However, Eclipse's constraint-based implementation does not foresee the problem and performs the refactoring without warning. ♦

One possible remedy for the problem of Example 3 is to add a constraint requiring an accessibility of `A.i` that makes it inaccessible from `C`, but then, this constraint makes no sense for the program before the refactoring — there is no `A.i` and why should accessibility of `B.i` be lowered? Again, a solution to this problem requires foresight of the situation after the refactoring.

Example 3 differs from Example 2 in that for the program as is, the field `B.i` is completely unrelated to the rest of the program, so that there seems to be no reason at all to generate a constraint for it (in fact, `B.i` could even be deleted without changing the meaning of the program). This was different for Example 2, in which `B.i` was already considered in the constraint generation process, only not in a manner that was still sufficient after the refactoring. However, both examples have in common that with the refactoring intent known to the constraint generator, it could be tweaked to insert the necessary constraints. This will be different for

EXAMPLE 4: Extending Example 1 to

```
class Z { int i = 0; }
class A extends Z { { assert this.i == 0; } }
class B extends A {
  int i = 1;
  int j = this.i;
}
```

the pulling up of `B.j` to `A` should be rejected since the necessary accompanying pulling up of `B.i` to `A` changes the binding of references to `i` on instances of type `A`, unless all receivers of references to `Z.i` of (static) type `A` (here: `this` in the assert statement) are cast to `Z`, or one of `Z.i` and `B.i` is renamed. Again, the current constraint-based implementation of PULL UP FIELD in Eclipse fails to see this. ♦

The problem highlighted by this example is that, given that the refactoring intent is to pull up `B.j`, it is difficult to foresee, for a constraint generator applied to the original program and refactoring intent, that access to `Z.i` should involve a cast, or a field should be renamed, since it is unknown, before the generated CSP has been solved, that `B.i` must be pulled up as well. Without foresight of that such a change might happen, no constraints protecting bindings to `Z.i` will be generated. Generally, refactorings can have far-reaching ripple effects that are difficult to foresee, and a correct implementation must account for them all. This is a non-trivial problem.

2.2 Related Work

In his doctoral dissertation, Griswold used bidirectional mappings between Scheme programs and program dependence graphs that allowed him to perform restructurings on the latter (and thus on a representation in which behaviour preservation is relatively easy to assert) [8]. This approach is somewhat analogous to that taken by constraint-based refactoring, which transforms a program to a CSP, in which refactoring amounts to constraint solving (see Section 3 for details).

Constraint-based type refactorings as pioneered by Tip et al. [19, 20] make a number of simplifying assumptions avoiding most problems that we are addressing here. In particular, they assume that program elements whose moving may accidentally change behaviour-critical dependencies (such as binding or overriding relationships) are adequately renamed before the move [20]. However, even if programs are prepared in such manner, the constraint rules provided in [20] still fail to address some of the foresight problems we are solving: for instance, rule 12 (for hiding) makes sure that existing hiding relationships are preserved (preventing a renaming), but cannot avoid a change of binding due to the pulling up of a field, as in our Example 4.

The work of Schäfer et al. [13–15] avoids accidental changes of behaviour-critical program dependencies by recording them prior to refactoring and by introducing a correction phase that restores the original dependencies, if possible, after the intended refactoring has been performed (solving the problem with hindsight, so to say). However, the changes necessary to perform the correction may themselves affect well-formedness and meaning of a program, which is why Schäfer's approach of locked dependencies has recently been combined with constraints [16]. As we will show, our work presented here is more general not only in that it is capable of addressing the primary refactoring and all corrections required using a single formalism (avoiding the looping between dependency locking/unlocking and constraint solving), but also in that it can choose between different measures for maintaining the original dependencies: for instance, it may rename problematic program elements, or make them inaccessible, or move them to locations in which they do not interfere.

Dynamically changing systems gave rise to the investigation of so-called *dynamic constraint satisfaction problems* [11], in which the activeness of certain constraints depends on the satisfaction of others. More specifically, the introduction of conditional constraints allows the constraint solver to explore dynamic reconfigurations of (usually hardware) systems certain components of which may or may not be present (i.e., switched on or off). This situation is not unlike the foresight problem of refactoring, which must also let dynamically changing configurations (that is, changes to the program structure) be explored by the constraint solver. However, restructuring software must deal with the more general problem of moving program elements around, and placing a switch at every possible location would be absurdly expensive (especially since, as has been shown in [7], the computational burden of conditional constraints is heavy).

In object-oriented programming, conditional constraints have been used for type inference [12] and also for certain constraint-based refactorings [1, 3]. In particular, [3] has used conditional constraints (there called *guarded constraints*) to handle the interplay of parameterized and raw types when converting Java programs to use generic libraries: whereas parameterized types require additional constraints on variables

representing the type parameters, raw types do not give rise to such variables which, consequently, cannot be constrained. Constraints generated from assignments must therefore be made sensitive to the "parameterizedness" of the participants and, since the parameterizedness may be changed by the refactoring, this sensitivity must be dynamic. Similarly, [1] has used conditional constraints (there called *implication constraints*) to let generated type constraints depend on a binary switch indicating whether an occurrence of a constructor or method call of a legacy type has been replaced (by the constraint solver) with an equivalent call of a migration type. However, both problems are analogous to the hardware problem of switching on or off components, whereas we have to deal with moving program elements to new locations.

In earlier work of ours on constraining accessibility under refactoring, we introduced so-called foresight application of constraint rules which, knowing to which location a program element was to be moved by a refactoring (the refactoring intent), computed the required accessibilities for that location [17]. However, since this computation occurred outside the constraint solving process, we had to know in advance which program elements were to be moved where, which, as argued above, is an unrealistic assumption in the general case.

3 A Brief Recap of Constraint-Based Refactoring

In constraint-based refactoring, a program is sufficiently represented by

- a set of variables, called *constraint variables*, representing selected properties of the program and
- a set of relationships, called *constraints*, constraining the properties, representing syntactic and semantic rules of the programming language as applied to the program.

Together, the constraint variables and the constraints define a CSP whose solution space represents programs that are refactorings of each other. For instance, the CSP corresponding to the refactoring problem of Example 1 consists of the constraint set

$$\{v_1 = v_2 = v_3,\ v_2 \leq v_4\} \tag{1}$$

where v_1 represents the declaring type of `j`, v_2 represents the location of `this`, v_3 represent the location of the reference to `i`, and v_4 represents the declaring type of `i` (all having the initial value `B`; note that (1) is solved with these values). Pulling up `j` translates to assigning v_1 the new value `A`, which (via the equality constraint $v_1 = v_2 = v_3$) is propagated to v_2, which in turn (via the inequality constraint $v_2 \leq v_4$) requires v_4 to change to `A` as well, translating to pulling up `i` along with `j`.

Generally, the CSP representing a refactoring problem is solved with the initial values of the constraint variables assigned. A refactoring intent (such as pulling up field `j` from `B` to `A`) translates to changing one or more variable values, which may require other variable values to change as well for the CSP to remain solved, which ones precisely being computed by a constraint solver. Each solution of the CSP then corresponds to a refactored program that is obtained by writing back the values of the changed variables to the original program.

Table 1. Properties of program elements used in more than one occasion throughout this paper

PROPERTY	MEANING
$e.\alpha$	the *declared accessibility* of e (corresponding to $\langle e \rangle$ in [17])
$e.\lambda$	the *location* of e, the type in whose body e occurs (corresp. to $\lambda(e)$ in [17] and, for declared entities, to *Decl*(e), the *declaring type* of e, in [19])
$e.\lambda_T$	the *top level type* hosting e; same as $e.\lambda$ for elements directly occurring in the bodies of top level types
$e.\iota$	the *identifier* of e
$e.\pi$	the *package* hosting e
$e.\tau$	the *type* of e (declared or inferred; corresponding to $[e]$ in [19])

3.1 Constraint Rules

A CSP such as (1) that represents a program to be refactored is generated from this program by application of so-called *constraint rules*, which are generally of the form

$$\frac{\textit{program queries}}{\textit{constraints}}$$

Here, *program queries* is a set of predicates (implicitly conjoined) that are interpreted as queries over a program, and *constraints* represents the set of constraints to be generated (added to the CSP) for program elements selected by the queries. Both the program queries and constraints contain variables which are bound to *program elements* (*declared entities* and *references* to declared entities of a program; the nodes of its AST) by the queries; the constraint rule is implicitly universally quantified over these variables (note that these variables are *not* the constraint variables).

EXAMPLE. Application of the constraint rule

$$\frac{\textit{overrides}(M_2, M_1)}{\textit{accessible}(M_2, M_1)}$$

to a program searches the program for occurrences of pairs of methods (M_2, M_1) such that M_2 overrides M_1, and generates for each found pair a constraint requiring that M_1 is accessible from M_2 ([6], §8.4.8.1). ♦

Constraints such as the above *accessible*(M_2, M_1) generated by the application of constraint rules do not constrain program elements directly — rather, they constrain *properties of the program elements* (which are therefore the constraint variables of the CSP). The properties of program elements and their domains depend on the elements' kinds (i.e., whether an element is a declared entity or a reference, whether a declared entity is a method, a field, etc.): for instance, the declaration of a field has at least the properties *location* (where the field is declared; the hosting type), *type* (the declared type of the field), and *accessibility* (the access modifier used in the declaration, in Java one of *private*, *package*, *protected*, or *public*). We use Greek letters to denote

properties: $e.\tau$ for the type of element e, $e.\lambda$ for the location of e, $e.\alpha$ for the declared accessibility, etc.; Table 1 summarizes the properties that we will be using repeatedly throughout this paper.

EXAMPLE. A spelled out and extended variant of the previous constraint rule is

$$\frac{\mathit{overrides}(M_2, M_1)}{M_2.\lambda \leq_\tau M_1.\lambda \quad M_1.\alpha \geq_\alpha \alpha(M_2, M_1)}$$

in which $\leq_\tau$ represents the subtype relationship defined by the program, $>_\alpha$ represents the (total) ordering of access modifiers in Java (with $\geq_\alpha$ being defined as usual), and α is a helper function computing the minimum required accessibility for the declared entity of the second argument when accessed from the location of the first ([17]; see Figure 2 for how α is defined in terms of constraints). Taken alone, these constraints allow it that M_1 or M_2 are moved up or down the class hierarchy as long as $M_2.\lambda$ remains a subtype of $M_1.\lambda$, and that the declared accessibility $M_1.\alpha$ may be increased or lowered, as long as it remains above what is required by the locations of M_1 and M_2 relative to each other. ♦

The constraint rules governing the PULL UP FIELD refactoring of Example 1 are

$$\frac{\mathit{same\text{-}statement}(e_1, \ldots, e_n)}{e_1.\lambda = \ldots = e_n.\lambda}$$

where the variable argument query *same-statement*$(e_1, \ldots, e_n)$ finds all tuples of program elements occurring in the same statement, and

$$\frac{\mathit{binds}(f, F) \quad \mathit{receiver}(f, r)}{r.\tau \leq_\tau F.\lambda}$$

where *binds* finds all pairs of field accesses f and field declarations F such that f binds to F, and *receiver* finds all pairs of field accesses f and references r such that r is the receiver of f.[1] Applied to the program of Example 1, these two rules generate the constraint set

$$\{\ \mathsf{j}^{\mathsf{B}}.\lambda = \mathsf{this}_{\mathsf{B}}.\lambda = \mathsf{i}_{\mathsf{B}}.\lambda,\ \mathsf{this}_{\mathsf{B}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda\ \}$$

(with j^{B} representing `B.j`, $\mathsf{this}_{\mathsf{B}}$ representing the reference to `this` in `B`, etc.).

It is instructive to note that to a certain extent, program queries and constraints can be exchanged for each other. For instance, the expression $e_1.\lambda = e_2.\lambda$ can be interpreted as a query, in which case it means "select all pairs of program elements (e_1, e_2) such that e_1 and e_2 are located in the same type", or interpreted as a constraint, meaning "whatever the location of e_1 or e_2, it must be the same as the other". The main differences are operational: whereas the query finds all instances of e_1 and e_2 in the program that satisfy the stated condition, the constraint makes sure that the properties of the found instances (representing the constraint variables) always remain aligned. Also, while program queries are evaluated at rule application (i.e., constraint generation) time, when all properties have their initial values, constraints are evaluated at constraint solution time, during which the values of the properties may be changed. This latter difference will play an important role below.

[1] Following the convention of [19], we use upper case letters for variables representing declared entities, and lower case letters for variables representing references.

3.2 Conditional Constraints

A *conditional constraint* [7, 11] of the form $P \rightarrow C$ is a constraint over two (reified) constraints, the *premise constraint*, P, and the *consequent constraint*, C. Satisfaction of $P \rightarrow C$ requires satisfaction of C only if P is satisfied; if not, C can be ignored. P can therefore be considered a guard switching C on or off. Conditional constraints are readily handled by contemporary constraint solvers (e.g., [2]).

Conditional constraints have many uses in constraint-based refactoring. For instance, the JLS mandates that if two fields are declared in the same statement, their declared type, τ, must be the same ([6], §8.3). Expressed as a constraint rule:

$$\frac{\textit{same-declaration}(F_1, F_2)}{F_1.\tau = F_2.\tau} \tag{2}$$

If, for some reason, a refactoring required that the declared type of one, but not both, of f_1 and f_2 is changed, the refactoring would have to be refused. However, this rejection may be overly strict, namely if the declaration can be split as part of the refactoring (in which case the constraint need no longer hold). The constraint rule

$$\frac{\textit{same-declaration}(F_1, F_2)}{F_1.\sigma = F_2.\sigma \rightarrow F_1.\tau = F_2.\tau} \tag{3}$$

in which the property σ represents the statement in which a field is declared, generates a conditional constraint that solves this problem: only if F_1 and F_2 are declared in the same statement need the types of F_1 and F_2 be the same. If the constraint solver can assign $F_1.\sigma$ or $F_2.\sigma$ a new value so that $F_1.\sigma \neq F_2.\sigma$, the declared types of F_1 and F_2 may differ. Thus, the constraint solver can compute that splitting the declaration solves the refactoring problem.

3.3 Specification of Refactorings

Constraint rules are generally independent of refactorings. However, not all constraint rules are applicable or relevant for all refactorings. This is so because not all properties may be changed by all refactorings: for instance, if the intended refactoring is to pull up a field, renaming that field or others that stand in the way of the pulling up may not be compatible with the refactoring intent, so that identifiers are fixed for this refactoring. Thus, the full specification of an intended refactoring (a refactoring problem) involves

- the program to be refactored,
- the set of constraint rules constraining the properties whose changes are associated with the refactoring,
- the concrete refactoring to be performed, as expressed by a selection of properties (usually one) and their new, mandatory values, and
- a specification of the other properties the constraint solver is allowed to change in order to perform the refactoring. [18]

The last item divides the properties extracted from a program into two kinds, those whose values are *fixed* and those whose values are *non-fixed*. This distinction will also play an important role in our treatment of the foresight problem.

4 Constraint Rule Rewriting

Ignoring the operational differences (noted at the end of Section 3.1) between the query *same-declaration*(F_1, F_2) and the constraint $F_1.\sigma = F_2.\sigma$, the two appear to express the same thing in different terms. In fact, considering that both a constraint rule and a conditional constraint are implications of some kind, (3) contains a tautology: either the query of the constraint rule or the premise of the conditional constraint could be dropped without affecting the contribution of the constraint rule to a refactoring. The only caveat is that the choice which one to drop is not free.

To see why this is the case, we have to look at the variability of program properties and the different evaluation times of queries and constraints. If all properties involved in the premise of a conditional constraint are known to be always fixed (i.e., their only allowed values are their initial values), satisfaction of the premise can be computed at rule application time (when the constraints are generated), after the variables in the queries have been instantiated with program elements. Thus, the premise can be pulled up ("promoted") to the rule precedent, transforming (3) to

$$\frac{\textit{same-declaration}(F_1, F_2) \quad F_1.\sigma = F_2.\sigma}{F_1.\tau = F_2.\tau}$$

Since $F_1.\sigma = F_2.\sigma$ as a query has the same meaning as *same-declaration*(F_1, F_2), the former can be dropped, giving us the simplified rule (2). This kind of rule rewriting is worthwhile since it saves the generation of conditional constraints.

If however the properties involved in the premise of a conditional constraint are non-fixed (so that their values may be changed by the solver), satisfaction of the premise cannot be computed at rule application time. In fact, in this case it is even questionable whether an equivalent query should be evaluated at this time, since this restricts the generation of the conditional constraint to program elements fulfilling the premise for the program as is. For instance, the query of (3) requires that F_1 and F_2 are declared in the same statement, so that no (conditional) constraint will be generated for pairs of fields that are not, preventing the solver from merging two field declarations separate at the time of constraint generation into one should their types (become) equal. In that case, the query should be pushed down ("demoted") to the premise of a conditional constraint, transforming (3) to

$$\frac{F_1 \quad F_2}{\textit{same-declaration}(F_1, F_2) \rightarrow F_1.\sigma = F_2.\sigma \rightarrow F_1.\tau = F_2.\tau}$$

or, since $F_1.\sigma = F_2.\sigma$ and *same-declaration*(F_1, F_2) as constraints are equivalent, to

$$\frac{F_1 \quad F_2}{F_1.\sigma = F_2.\sigma \rightarrow F_1.\tau = F_2.\tau}$$

Generally, if the precedent of a constraint rule contains queries that relate to properties of the program that may change during the refactoring, those queries (rephrased as constraints) should be pushed down from the rule precedent to the premise of a conditional constraint in the consequent of the rule. This *demotion of queries* to premises of conditional constraints serves the generalization of constraint rules (so that more constraints that cover more refactoring problems are generated). Conversely, if the premise of a conditional constraint in a rule consequent can always be evaluated at rule application time (since for every application it constrains only fixed properties), the premise can be pulled up to the rule precedent. This *promotion of premises* of conditional constraints to queries of constraint rules serves the tuning of constraint-based refactoring, by making constraint generation more specific (so that, if the promoted queries are not redundant to existing queries, fewer constraints are generated), and by simplifying the constraints that must be solved. Both promotion and demotion will be made use of in our solution of the foresight problem as presented in Section 6.

5 Quantified Constraints

As pointed out in Section 3.1, constraint rules are implicitly universally quantified over the elements of a program. However, every single application of a constraint rule generates only one instance of the constraints in its consequent. There are situations in which this is insufficient, as demonstrated by the following

EXAMPLE: Beginning with Java 5, a method may be annotated with the `@Override` annotation, in which case the compiler checks that the method overrides a method defined by a superclass. This is captured by the constraint rule

$$\frac{\mathit{overrides}(M)}{\exists\, M' \neq M : M.\lambda <_\tau M'.\lambda \wedge M'.\alpha \geq_\alpha \alpha(M, M') \wedge \mathit{override\text{-}equivalent}(M', M)}$$

which requires that there is at least one method defined in a superclass that is accessible from M and has an override-equivalent signature ([6], §8.4.2). ♦

Unlike conditional constraints, quantified constraints are not readily handled by available constraint solvers. However, since the domains that are being quantified over, namely sets of program elements, are always finite, a quantified constraint can be unrolled to a finite disjunction or conjunction of constraints. For instance, if a program has three methods, M_1, M_2, and M_3, of which M_1 is annotated with `@Override`, application of the above constraint rule unrolls to

$$\begin{aligned} &M_1.\lambda \leq_\tau M_2.\lambda \wedge M_2.\alpha \geq_\alpha \alpha(M_1, M_2) \wedge \mathit{override\text{-}equivalent}(M_2, M_1) \\ \vee\; &M_1.\lambda \leq_\tau M_3.\lambda \wedge M_3.\alpha \geq_\alpha \alpha(M_1, M_3) \wedge \mathit{override\text{-}equivalent}(M_3, M_1) \end{aligned}$$

Contrasting this simple one, below we will encounter examples of quantified constraints whose unrolling is exceedingly expensive.

Quantified constraints also offer opportunities for rule rewriting. Because constraint rules are implicitly universally quantified over their variables, a universally quantified constraint occurring in a rule consequent can be stripped of the quantifier, by moving the quantified variable (representing program elements) to the rule precedent. Effectively, this makes unrolling a universally quantified constraint an immanent part of constraint generation (rule application). This "promotion" of universal quantification will be exploited by our capture of foresight, as detailed in Section 6.

EXAMPLE. The JLS mandates that of all top-level classes contained in a compilation unit, only one may be declared *public*. This translates to the constraint rule

$$\frac{\textit{top-level-class}(C) \quad C.\alpha = \textit{public}}{\forall C' \neq C,\ \textit{top-level-class}(C'):\ C'.\upsilon = C.\upsilon \rightarrow C'.\alpha <_\alpha \textit{public}}$$

(in which $C.\upsilon$ represents the compilation unit of C), which is equivalent to

$$\frac{\textit{top-level-class}(C) \quad C.\alpha = \textit{public} \quad C' \neq C \quad \textit{top-level-class}(C')}{C'.\upsilon = C.\upsilon \rightarrow C'.\alpha <_\alpha \textit{public}}$$

in which the explicit universal quantification in the consequent has been replaced by the introduction of C' as a variable in the rule precedent whose quantification (and unrolling) is implicit in rule application. If the intended refactoring does not allow moving classes between compilation units, the rule can be further rewritten to

$$\frac{\textit{top-level-class}(C) \quad C.\alpha = \textit{public} \quad C' \neq C \quad \textit{top-level-class}(C') \quad C'.\upsilon = C.\upsilon}{C'.\alpha <_\alpha \textit{public}}$$

(by promoting the premise of the conditional constraint to the rule precedent), saving the generation of conditionals for classes of the same compilation unit, and the generation of constraints for classes from different compilation units altogether. ♦

6 A Constraint-Based Solution of the Foresight Problem

The foresight problem exposed by Examples 2–4 of Section 2.1 is that the constraints generated from the constraint rules as applied to the program as is are insufficient: certain constraints are missing. With constraint rule rewriting and quantified constraints at hand, we are sufficiently equipped to systematically generate them.

6.1 Foresight with Constraint Rule Rewriting

Example 2 of Section 2.1 suggests that a constraint should have been generated that constrains the declared accessibility of field `i` to `public` *after* its pulling up to a class of another package, a constraint that would however have constrained the program as is incorrectly. Generation of a conditional constraint escapes this dilemma, by guarding the constraint with the condition that the access occurs from another package via a reference whose (static) type is not a (non-strict) supertype of the declaring type of `i`. This is obtained by rewriting the constraint rule of Example 2, here formalized as

$$\frac{\textit{binds}(m, M) \quad \textit{receiver}(m, r) \quad m.\pi \neq M.\pi \quad \neg\, r.\tau \leq_\tau r.\lambda}{M.\alpha = \textit{public}}$$

to

$$\frac{\textit{binds}(m, M) \quad \textit{receiver}(m, r)}{m.\pi \neq M.\pi \wedge \neg\, r.\tau \leq_\tau r.\lambda \rightarrow M.\alpha = \textit{public}} \tag{4}$$

in which the queries $m.\pi \neq M.\pi$ and $\neg\, r.\tau \leq_\tau r.\lambda$ have been demoted to the guard of a conditional constraint. Applied to the program of Example 2, this rule generates a consequent constraint ($M.\alpha = public$) that is inactive (switched off) for the program as is; if however the program is changed in such a way that the guard holds, the consequent is activated, and contributes to the refactoring. Thus, the rewritten rule codes foresight of the possible change.

As can be seen the use of constraint rules with demoted queries is expensive in that it leads to the generation of more constraints, and conditional ones at that. Therefore, if rule (4) is used in a specific refactoring, say GENERALIZE TYPE [19], fixed constraints should be promoted to queries, in the case of GENERALIZE TYPE leading to

$$\frac{binds(m, M) \quad receiver(m, r) \quad m.\pi \neq M.\pi \quad M.\alpha \neq public}{r.\tau \leq_\tau r.\lambda}$$

if only the declared types of program elements may be changed by the refactoring.

6.2 Foresight with Quantified Constraints

As detailed in Section 2.1, the problem highlighted by Example 3 is of a different nature than that of Example 2 in that a declared entity must be constrained that, for the program as is, is unrelated to the program elements to be refactored. This lack of relatedness suggests that such program elements escape ordinary constraint rules.

This is where quantified constraints step in. For the case of Example 3, that no field must exist to which a reference could bind alternatively (so that the reference would be ambiguous) is conveniently expressed using a non-existence constraint in the consequent of a constraint rule, as in

$$\frac{binds(f, F) \quad receiver(f, r)}{\neg\exists\, F' \neq F : F'.\iota = f.\iota \wedge F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\, F.\lambda <_\tau F'.\lambda} \tag{5}$$

which reads "there must not exist a field F' distinct from F that has the same name (identifier) as f (the reference that must not be ambiguous), that is accessible for f, that is declared in a supertype of the type of receiver r, and that is not declared in a supertype of the declaring type of F". Similarly, the problem exposed by Example 4, namely that no field must exist that hides the field a reference currently binds to, is countered by generating the constraint

$$\neg\exists\, F' \neq F : F'.\iota = f.\iota \wedge r.\tau \leq_\tau F'.\lambda \wedge F'.\lambda <_\tau F.\lambda$$

Note that in both cases, all conjuncts of the quantified constraint but the last equally apply as conditions required for f to bind to F — to avoid ambiguity or rebinding, conditions sufficient for f binding to F must not hold for other fields F' as well.

Unlike in the example of Section 5, unrolling quantified constraints such as the above can be very expensive. In the worst case, if a refactoring may change the name of a field and its location freely, a constraint must be generated for every other field in the program, and with it for every reference to a field (which may have to be renamed as well). However, most refactorings are not granted this freedom, so that constraint rules such as the above (which directly mirror the rules of the programming language) can be rewritten to suit specific refactorings.

Generally, constraint rules expressing that no program element must exist with properties that would infringe the program's well-formedness or change its behaviour have the form

$$\frac{query(e, \ldots)}{C(e, \ldots) \quad \neg\exists\, e' \neq e : C'(e', \ldots)} \tag{6}$$

in which $C(e, \ldots)$ and $C'(e', \ldots)$ represent arbitrary constraints expressing the relationships between the properties of e and others that must hold, and relationships between the properties of e' and others that must not hold. In a first rewriting step, we split the constraint rule (6) into two

$$\frac{query(e, \ldots)}{C(e, \ldots)} \qquad \frac{query(e, \ldots)}{\neg\exists\, e' \neq e : C'(e', \ldots)}$$

and leave aside the first, since it is standard. Next, we split the quantified constraint $C'(e', \ldots)$ into two conjuncts $F(e', \ldots)$ and $N(e', \ldots)$, the former containing only constraints whose constrained properties (constraint variables) are fixed for the refactoring, the latter containing the constraints of which at least one constrained property is non-fixed (note that either conjunct may be empty). This lets us rewrite the rule to

$$\frac{query(e, \ldots)}{\neg\exists\, e' \neq e : F(e', \ldots) \wedge N(e', \ldots)}$$

which is equivalent to

$$\frac{query(e, \ldots)}{\forall\, e' \neq e : \neg(\, F(e', \ldots) \wedge N(e', \ldots)\,)}$$

which is in turn equivalent to

$$\frac{query(e, \ldots)}{\forall\, e' \neq e : F(e', \ldots) \rightarrow \neg N(e', \ldots)}$$

Since the rule consequent is now a universally quantified conditional constraint whose premise depends on fixed properties only, we can rewrite the rule to

$$\frac{query(e, \ldots) \quad F(e', \ldots)}{\neg N(e', \ldots)}$$

whose additional query $F(e', \ldots)$ acts as a filter leading to the generation of fewer constraints than would have been introduced by the unrolling of $\neg\exists\, e' \neq e$: $C'(e', \ldots)$. For instance, for a refactoring that is not allowed to change identifiers or declared accessibilities, rule (5) can be rewritten to

$$\frac{binds(f, F) \quad receiver(f, r) \quad F' \neq F \quad F'.\iota \neq f.\iota}{F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\, F.\lambda < F'.\lambda} \tag{7}$$

Note that the conjunct constraining accessibility cannot be promoted to a query, since it depends on location (cf. the definition of α in Figure 2), which may be changed by the refactoring (but see below for further savings possible).

Generally, what seems like a rather discouraging threat to the tractability of constraint-based refactoring with foresight may be tamed by rule rewriting, allowing the evaluation of constraints — as queries — at rule application time. How effective this is in practice will be explored in the evaluation of Section 8.

6.3 Further Savings

The above introduced possible rewritings of constraint rules depend on the (lack of) variability of constrained properties in the rule consequent, more specifically on whether the satisfaction of a constraint can be decided — for every possible application of the rule — at rule application time: if it can, the constraint can be promoted to a query where it acts as a filter causing fewer generated constraints. This lets us tailor general (i.e., refactoring-independent) constraint rules to a specific refactoring characterized by which properties are non-fixed and which are fixed (cf. Section 3.3). This tailoring can be carried out before the refactoring is actually applied, as it holds across all possible applications. Practically, this means that it can be performed for the tuning of a set of constraint rules to a specific refactoring tool.

However, further savings are possible when a (specifically tailored) refactoring is actually performed. When a constraint rule is applied, i.e., when all variables of the rule have been instantiated with concrete program elements, it may be the case that individual constraints to be generated can already be evaluated. For instance, continuing the example of rewriting rule (5) from the previous subsection to rule (7), for all fields F' whose declared accessibility is *public*, the constraint $F'.\alpha \geq_\alpha \alpha(f, F')$ in the consequent of (7) need not be generated, since it is always satisfied (recall that the refactoring was not allowed to change accessibility). Furthermore, reified constraints (cf. Section 3.2) that can be evaluated at rule application time may lead to shortcut evaluations of the Boolean constraints (including conditional constraints) constraining them, which may lead to further savings (including immediate abortion of a refactoring if a top-level constraint is unsatisfiable). These optimizations do not correspond to rewritings of constraint rules, since they are performed individually, for single applications of rules. We will evaluate their impact in Section 8.

6.4 Basic Algorithm of Constraint-Based Refactoring with Foresight

An algorithm for constraint-based refactoring with foresight that performs the possible tailoring described in Sections 6.1 and 6.2 and the additional optimizations of Section 6.3 is shown in Figure 1. It takes as input the parameters necessary to specify a constraint-based refactoring (as detailed in Section 3.3) and produces a refactored program, if the refactoring can be performed.

The algorithm is split into four stages: the rewriting (tailoring) of the constraint rules to the specific refactoring, the application of the rules to the program to be refactored, the performing of the individual optimizations, and the generation and solution of a CSP (including writing back the solution of the CSP to the original program). Some explanations follow:

Algorithm *RefactoringWithForesight*(*P*, *R*, *I*, *F*)
Input:
P, the program to be refactored
R, a set of constraint rules
I, the refactoring intent (a set of properties and their target values)
F(*p*), a filter selecting the non-fixed properties *p*
Output:
C, a CSP
P, the refactored program
Steps:
rule rewriting
1. **for each** constraint rule *r* **in** *R*
2. convert the rule consequent of *r* to disjunctive normal form (DNF)
3. extract the disjuncts of the DNF that are filtered as invariant
4. promote the extracted disjuncts to negated conjuncts of the rule precedent
5. **if** the remainder (variable disjuncts) is not empty, make it the new consequent
6. **else** drop the rule for this refactoring

rule application (constraint generation)
7. **for each** constraint rule *r* transformed as above
8. apply *r* to *P*, by evaluating the program queries and promoted constraints
9. **for each** match, instantiate the constraints of *r*'s consequent

early evaluation
10. **for each** instantiated constraint *c*
11. **if** any of its disjuncts evaluates to *true*, drop the entire constraint
12. **else if** all disjuncts evaluate to *false*, **fail**
13. **else** delete the disjuncts evaluating to *false* from the constraint and add it to *C*

initializing and solving the CSP, and writing back
14. **for each** property *p* **in** *P* constrained by a constraint in *C*
15. initialize *p* with its value from *P*
16. **if** $p \in I$, replace its initial value with that in *I*
17. **if** $F(p) \wedge p \notin I$, set *p*'s domain according to the type of *p*
18. **else** make *p* constant
19. **if** *C* is solvable
20. solve *C*
21. **for each** changed property *p* **in** *P* write back its new value to *P* to reflect the change
22. **else fail**

Fig. 1. Basic algorithm of refactoring with foresight

Step 2: Conversion to DNF is performed after replacing $\neg\exists\, x\colon \varphi(x)$ with $\forall\, x\colon \neg\varphi(x)$ and dropping the explicit universal quantification as shown in Section 4.

Step 3: A disjunct is filtered as invariant if none of its constrained properties may be changed (meaning that its satisfiability depends only on its instantiation during rule application in Step 9). Note that since, at this stage, all properties are properties of unbound variables (the rules have not yet been applied to actual program elements), the filtering condition must hold for all program elements that can be substituted for the variables. More specifically, only filters of the kind "all access modifiers may be changed" can be evaluated at this stage.

Step 4: A set of invariant disjuncts $A_1, \ldots, A_n$ with (variant) remainder B (which may itself be a disjunction) is interpreted as the precedent $A := \neg(A_1 \vee \ldots \vee A_n)$ of

a conditional constraint $A \rightarrow B$, which, as explained in Section 4, can be promoted to the rule precedent (a conjunction of queries), where it is added as $\neg A_1 \wedge \ldots \wedge \neg A_n$.

Step 6: If none of the constrained properties are changeable, the constraint adds nothing to the solution (recall that all constraints are always satisfied initially).

Step 8: Evaluation of the program queries and promoted constraints substitutes the variables of the rules with the program elements matching the queries.

Step 11: Since the constraints c are in DNF, one disjunct evaluating to *true* renders all others irrelevant. For a disjunct to evaluate to *true* at this stage (i.e., before the actual constraint solving), the values of the constrained properties must be invariant; their value is then the initial value (obtained as in Step 15).

Step 12: Since all constraints of a CSP are implicitly conjoined, there is no chance of a solution if a single constraint always (under all assignments) fails.

Step 13: There is a third case since not all disjuncts can always be evaluated at this stage: those whose constrained properties are at least partly variable (as decided by F) depend on values assigned by the solver (Step 20).

The effectiveness of the savings introduced by algorithm *RefactoringWithForesight* depends on the number of disjuncts in the rule consequents (as introduced by the conditional constraints expressing foresight) and on the selectivity of the filter F (Step 3). In particular, if the filter F selects many properties as non-fixed (meaning that many different kinds of changes are allowed), opportunities for rule rewriting are rare. However, in these cases early evaluation may still be effective, especially if only some properties of a specific kind (such as locations of fields) are non-fixed (as is typically the case for filters such as "allow only locations of fields of the same class to change", a filter used by the PULL UP FIELD refactoring). Our evaluation in Section 8 will shed light on the effectiveness of rule rewriting and early evaluation.

7 Implementation

We have implemented refactoring with foresight as described here as an extension to our refactoring constraint language REFACOLA [18]. REFACOLA allows the developer of a refactoring tool to define different *kinds* of program elements (beyond the declared entity and reference distinction made in this paper, e.g., `Field`, `Method`, `Variable`, etc.), and to associate with each kind a fixed set of *properties* (such as the ones listed in Table 1). Each property comes with a *domain*, which may be predefined (such as `Identifier`), enumerated (such as `Accessibility`), or program-dependent (such as `Location`). The REFACOLA language is complemented by a REFACOLA framework which provides a predefined set of program queries, a generic algorithm for applying the constraint rules to a program, an interface to constraint solvers such as Choco [2], and routines for writing back the solved constraints to the program source. The REFACOLA compiler and framework have been implemented as plugins to Eclipse, with adapters for C# and Eiffel compilers. Refactoring specifications in REFACOLA are completely declarative: refactoring tools can be generated from these specifications at the push of a button. The generated tool used for the evaluation in Section 8 (enhanced with a basic user interface) can be downloaded from `www.feu.de/ps/prjs/refacola`.

One of the main contributions of the REFACOLA framework is its *GenerateConstraints* algorithm [18], which keeps constraint-based refactoring tractable by generating only the constraints constraining (properties of) program elements that are, directly or indirectly, related to the code change intended by the refactoring (so that the change can propagate to them). In our current work, the savings achieved by this algorithm appear to be traded for addressing the foresight problem, since quantified constraints providently involve all program elements of a given kind, including ones seemingly unrelated to the refactoring intent (cf. Example 3). However, as we will show next, the promotion of constraints to queries and the early evaluation of constraints allow us to retain much of the original savings in many cases.

8 Evaluation

To be able to judge the impact our solution to the foresight problem has on the viability of constraint-based refactoring in practice, including the effectiveness of the rule rewritings and early evaluation suggested, we have performed a systematic evaluation on the basis of several variants of the PULL UP FIELD refactoring [4] used as an example throughout this paper. We chose PULL UP FIELD because it strikes a good balance between simplicity of the refactoring (so that our focus is not diffused by other problems of refactoring) and occurrence of foresight problems (as suggested by the motivating examples). To be able to assess the impact rule rewriting and early evaluation have on constraint generation, and also the dependence on the permissiveness of the filter F (cf. Figure 1), we evaluated several variants of PULL UP FIELD that differ in the degrees of freedom granted to the refactoring, i.e., whether it is allowed to pull up other fields as well, rename fields, or change their accessibility.

8.1 Specification of PULL UP FIELD with Foresight

The constraint rules immediately relevant for PULL UP FIELD with foresight are shown in Figure 2. The program queries are given expressive names that serve to name the rules also (note how FIELDACCESS(r, f, F) combines *binds*(f, F) and *receiver*(r, f)); their implementation is of no interest here. We have omitted some general rules for enforcing well-formedness of locations (every nested type residing in a top-level type resides in the package the top-level type resides in; every program element residing in a type resides in the top-level type and package the type resides in; etc.), for restricting the accessibility of top-level types (only *package* and *public* are allowed) and members of interfaces (all *public*), etc.

The rules of Figure 2 are explained as follows: FIELDDECLARATION(F) requires that no two fields exist in the same class that have the same name (note that both identifier and location are considered non-fixed by this rule, and all others for that matter). INITIALIZINGFIELDDECLARATION(F, r) adds to it that each field F and the reference r that is assigned to it reside in the same location (*co-location*) and that the (inferred) type of the reference is a non-strict subtype of the declared type of the field (*typing*). THISACCESS(t) is the standard type inference rule for *this*, expressing that the type of *this* (as a reference) is the type it is located in.

FIELDDECLARATION(F)

$$\neg\exists\, F' \neq F : F'.\iota = F.\iota \wedge F'.\lambda = F.\lambda \quad \text{(no name collision)}$$

INITIALIZINGFIELDDECLARATION(F, r)

$$F.\lambda = r.\lambda \quad \text{(co-location)}$$
$$r.\tau \leq_\tau F.\tau \quad \text{(typing)}$$

THISACCESS(t)

$$t.\tau = t.\lambda \quad \text{(typing)}$$

FIELDACCESS(r, f, F)

$$f.\iota = F.\iota \quad \text{(name equality)}$$
$$f.\tau = F.\tau \quad \text{(typing)}$$
$$r.\lambda = f.\lambda \quad \text{(co-location)}$$
$$r.\tau \leq_\tau F.\lambda \quad \text{(member)}$$
$$F.\alpha \geq_\alpha \alpha(f, F) \quad \text{(accessible member)}$$
$$f.\pi \neq r.\tau.\pi \rightarrow r.\tau.\alpha = \mathit{public} \quad \text{(implicit type access)}$$
$$r.\tau < F.\lambda \rightarrow F.\alpha > \mathit{private} \quad \text{(inherited member access 1)}$$
$$\forall\, T : r.\tau <_\tau T \leq_\tau F.\lambda \rightarrow (F.\alpha <_\alpha \mathit{protected} \rightarrow T.\pi = F.\pi) \quad \text{(inherited member access 2)}$$
$$f.\pi \neq F.\pi \wedge \neg\, r.\tau \leq_\tau r.\lambda \rightarrow F.\alpha = \mathit{public} \quad \text{(protected accessibility)}$$
$$\neg\exists\, F' \neq F : F'.\iota = F.\iota \wedge F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\, F.\lambda <_\tau F'.\lambda \quad \text{(no ambiguity)}$$
$$\neg\exists\, F' \neq F : F'.\iota = F.\iota \wedge r.\tau \leq_\tau F'.\lambda \wedge F'.\lambda <_\tau F.\lambda \quad \text{(no hiding)}$$

$e_2.\alpha \geq_\alpha \alpha(e_1, e_2) \equiv$
if $e_1.\lambda_T = e_2.\lambda_T$ then $e_2.\alpha \geq_\alpha \mathit{private}$
else if $e_1.\pi = e_2.\pi$ then $e_2.\alpha \geq_\alpha \mathit{package}$
else if $e_1.\lambda \leq_\tau e_2.\lambda$ then $e_2.\alpha \geq_\alpha \mathit{protected}$
else $e_2.\alpha \geq_\alpha \mathit{public}$

Fig. 2. Constraint rules for the PULL UP FIELD refactoring (excerpt)

The rule FIELDACCESS(r, f, F) requires that: each field F and all references f to it have the same name (*name equality*); that the inferred type of f is the declared type of F (*typing*); that the type of the receiver r is a subtype of the declaring type of F (so that r has f as a member; *member*); that F is accessible from the location of f (*accessible member*); that the receiver type is accessible for f (*implicit type access*); that an inherited member cannot have *private* accessibility (*inherited member access 1*) and must have *public* accessibility if any intervening class on the inheritance path resides in a different package than F; and includes rule (4) of Section 6.1 (*protected accessibility*), as well as the rules from Section 6.2 (*no ambiguity* and *no hiding*).

Finally, Figure 2 shows how the function α is expressed as a nested conditional constraint, specifying how required accessibility of e_2 adapts to changes of location of e_1 and e_2 relative to each other.

As can be seen from Figure 2, the constraint rules FIELDDECLARATION and FIELDACCESS introduce three negated existential quantifications which, given that they are applied to each field declaration and field access of a program and that each one needs to be unrolled to all fields declared in the program, must be expected to lead to substantial numbers of additional constraints. This is countered by the

Table 2. Filters used for specifying the different variants of the PULL UP FIELD refactoring used in the evaluation

FILTER	DEFINITION (SPECIFYING NON-FIXED PROPERTIES)
NOC	$F_I.\lambda$, $F_I.\lambda_T$, and $F_I.\pi$, where F_I is the field to be pulled up
CL	$F.\lambda$, $F.\lambda_T$, and $F.\pi$, where F is all fields of the class of the field to be pulled up
CA	$F_I.\lambda$, $F_I.\lambda_T$, $F_I.\pi$, $F.\alpha$, and $T.\alpha$, where F is any field and T is a (its) type
CI	$F_I.\lambda$, $F_I.\lambda_T$, $F_I.\pi$, $F.\iota$, and $f.\iota$, where F is any field and f is any reference to a field
CLAI	CL ∪ CA ∪ CI

RefactoringWithForesight algorithm of Figure 1, whose rewriting stage, applied for instance with the filter "allow only locations of fields of class *C* to change" (the filter CL of Table 2), transforms the two constraint rules to

$$\frac{\text{FIELDDECLARATION}(F) \qquad F' \neq F \quad F'.\iota = F.\iota}{F'.\lambda \neq F.\lambda} \quad \text{(no name collision)}$$

$$\frac{\text{FIELDACCESS}(r, f, F) \qquad F' \neq F \quad F'.\iota = F.\iota}{\begin{array}{r}\ldots \\ F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\, F.\lambda <_\tau F'.\lambda \quad \text{(no ambiguity)} \\ \neg\, r.\tau \leq_\tau F'.\lambda \vee \neg\, F'.\lambda <_\tau F.\lambda \quad \text{(no hiding)}\end{array}}$$

Note that, had the filter been different (for instance, "allow only changes of identifiers"), the transformation would have been different. Also note that, although it is clear from the filter that not all properties λ of a program may be changed by the refactoring (only those of the same class), no further constraints from the rule consequents can be promoted to the precedent, since the rule applies to all properties of all elements of a program. This is different for the early evaluation stage of the algorithm which, when applied to the program of Example 4, generates the constraint set

$$\{\ \mathsf{i}^{\mathsf{Z}}.\lambda \neq_\tau \mathsf{i}^{\mathsf{B}}.\lambda,\ \mathsf{i}^{\mathsf{B}}.\lambda \neq_\tau \mathsf{i}^{\mathsf{Z}}.\lambda,\ \mathsf{this}_{\mathsf{B}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda,\ \neg\, \mathsf{i}^{\mathsf{Z}}.\lambda <_\tau \mathsf{i}^{\mathsf{B}}.\lambda,\ \neg\, \mathsf{this}_{\mathsf{A}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda \vee \neg\, \mathsf{i}^{\mathsf{B}}.\lambda <_\tau \mathsf{i}^{\mathsf{Z}}.\lambda\ \}$$

in which $\mathsf{this}_{\mathsf{A}}$ and $\mathsf{this}_{\mathsf{B}}$ refer to the references to this in classes A and B, respectively, and i^{Z} and i^{B} to the different declarations of i. As can be seen, these constraints (correctly) prevent the pulling up of i from B to A, since this would make the first disjunct of the last constraint ($\neg\, \mathsf{this}_{\mathsf{A}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda$) *false* without making the second disjunct (which is *false* with the program as is) *true*.

8.2 Variants of PULL UP FIELD

For our evaluation, we defined the PULL UP FIELD refactoring with five different degrees of freedom, based on the definition of the filters described in Table 2:

- Filter NOC (for *n*o *o*ther *c*hanges) specifies the basic variant of PULL UP FIELD: it excludes the automatic pulling up of other fields (as required by Examples 1 and 4) and the automatic adaptation of accessibilities (required by Examples 2 and 3).
- Filter CL (for *c*hange *l*ocation) enables the automatic pulling up of all other fields of the same class to the same target class.

Table 3. Sample projects used as the basis of the evaluation

PROJECT	NO. OF CLASSES	NO. OF FIELDS	PULL-UP OPPORTUNITIES†
ANTLR V3.2	71	118	221
Apac. commons.codec V1.3	19	56	210
Apache commons.io V1.4	74	47	80
Apache math V2.1	178	532	1164
Cream V1.06	32	63	71
Fit V1.1	95	122	237
HTML Parser V1.6	148	275	636
Jaxen V1.1.1	167	142	359
Jester V1.2.2	30	39	41
Junit V3.8.1	105	104	234
PicoContainer V1.3	73	116	348
total	992	1614	3601

† one per field and non-library superclass of the class declaring that field

- CA (for *c*hange *a*ccessibility) adjusts the accessibility of the pulled up field and of its type, if necessary. Any access modifier can be used.
- CI (*c*hange *i*dentifier) allows the refactoring to rename the pulled up field, or the field with the same name, to a fresh one in case of name collision, ambiguity, or hiding.
- CLAI (*c*hange *l*ocation, *a*ccessibility, or *i*dentifier) allows all additional changes.

Note that all of the above variants of PULL UP FIELD are completely defined by specifying the corresponding filter F that is supplied to the *RefactoringWithForesight* algorithm of Figure 1, and by supplying the domains of the properties selected by F (cf. [18]). Of the filters, NOC is the least permissive (allowing the fewest applications of the refactoring, because no other changes can be computed that would make the refactoring possible), and CLAI is the most permissive filter (allowing the most refactorings).

8.3 Experimental Setup and Results

To systematically evaluate our approach, we have applied our implementation of the algorithm of Figure 1 using the ruleset of Figure 2 to the sample programs of Table 3, using the following procedure:

```
let R be the constraint rules of Figure 2
for each sample program P of Table 3
  for each field f and class C of P such that f.λ<τC (ie, f is defined in a subclass of C)
    let I = { f.λ = C } (ie, pull up f to C)
    for each filter F of Table 2
      for each mode m of Table 4
        measure performance of RefactoringWithForesight(P, R, I, F) in mode m
```

The results of this procedure are shown in Table 5. Note that solvability and number of solutions are the same for modes f^{+}, f^{++}, and f^{+++}: this is so because rule rewriting and early evaluation are optimizations that have no effect on the solution space.

Table 4. Modes of application

MODE	FUNCTION
f^-	all foresight problem related constraints disabled
f^+	all foresight constraints, but no rewriting or constant evaluation, enabled
f^{++}	rule rewriting enabled
f^{+++}	early evaluation of constraints enabled

Ignoring foresight problems (mode f^-), PULL UP FIELD is almost always applicable (meaning that the corresponding CSP is solvable), even with no other changes allowed (filter NOC). Its applicability can only slightly be increased by allowing the refactoring to pull up other fields as well (filters CL and CLAI). The picture is entirely different when the constraints covering foresight problems are added (modes f^+, f^{++}, f^{+++}): with no other changes allowed (NOC), applicability is reduced by more than one half (64%). However, this reduced applicability (which prevents the refactoring from producing ill-formed or behaviourally changed programs) can be fully compensated by allowing PULL UP FIELD to make additional changes: with filter CLAI, all foresight problems can be solved by adapting locations, accessibilities, or identifiers, resulting in the exact same applicability (97.3%).

Of the additional changes permitted by CLAI, changing accessibility (filter CA) makes the biggest single contribution: taken alone, it improves applicability by 55.5 percent points. Note that by comparison, changing only identifiers (CI, which is equivalent in effect to Schäfer's name unlocking via insertion of qualifiers [13]) enables only a small fraction of refactorings affected by foresight problems. That allowing PULL UP FIELD to change locations of other fields (CL) increases its applicability in the foresight modes more than it does in f^- is due to the fact that in the former, pulling up a private field that is used to initialize another field requires pulling up the other field as well, if accessibility cannot be changed. This is ignored in mode f^-.

With respect to the cost introduced by addressing the foresight problem, Table 5 shows that without further measures (mode f^+), the rise in the number of constraints generated is dramatic: it 1200-folds on average. As was to be feared, the added constraints linking seemingly unrelated program elements to the refactoring intent reduce the effectiveness of the *GenerateConstraints* algorithm presented in [18] (cf. Section 6.4). However, as can also be seen from Table 5, tailoring constraint rules to a specific refactoring (here: to a specific variant of PULL UP FIELD as represented by a corresponding filter) via rule rewriting and applying early evaluation can reduce the number of constraints substantially: for NOC, CL, and CA it cuts the rise to less than 10-fold. For CI, rule rewriting is not as effective, however: this is so because for the expensive quantified constraints *no name collision*, *no hiding*, and *no ambiguity*, (cf. Figure 2) the constraint $F'.\iota = F.\iota$ cannot be promoted to a query (cf. Section 6.2), as which it would by highly selective (i.e., preclude the generation of many constraints; the number of fields with the same name in a program is usually small when

Table 5. Results of application to the pullable fields of Table 3 (all numbers averaged)

METRIC	MODE		FILTER				
			NOC	CL	CA	CI	CLAI
Solvable	f^-		97.0%	97.3%	97.0%	97.0%	97.3%
	f^+, f^{++}, f^{+++}		35.3%	36.6%	92.1%	38.7%	97.3%
No. of Constraints	f^-		10	36	10	10	36
	f^+		16694	16884	16740	17201	17456
	f^{++}		52	75	75	14227	17452
	f^{+++}		28	69	73	8816	17452
No. of Solutions	f^-		1	1.19	1	1	1.19
	f^+, f^{++}, f^{+++}		1	1.19	2.33	1.09	3.22
Times Required [ms][†]	f^-	gen	3	330	6	10	339
		solv	1	38	1	2	37
	f^+	gen	1251	2572	1807	1325	3191
		solv	69	98	70	94	160
	f^{++}	gen	907	1522	1587	5080	3085
		solv	1	26	4	75	164
	f^{+++}	gen	343	1429	1542	2912	3058
		solv	0	24	3	0	75

[†] all times obtained on contemporary laptops with 2 GHz clock speed running the Windows XP operating system with JVM heap space set to 1 GB, using the Choco [2] constraint solver

compared to the total number of fields). The additional constraints are approximately halved by early evaluation, which can exploit the fixedness of most other properties. However (and as was to be feared), average numbers suggest that both measures remain ineffective for the most permissive filter, CLAI: in this case, almost no rule rewritings and early evaluations seem to be possible.

On average, the number of solutions for all investigated variants of PULL UP FIELD remains within a range that would allow the user of the refactoring to inspect all alternatives and pick the one that is closest to his intent. Somewhat surprisingly, this is also the case for the foresight modes: one might have expected that covering more program elements' properties would lead to considerably more solutions. However, almost all constraints generated for filters NOC, CL, and CI are equality constraints (equality of locations or names) or disequality constraints with binary domains (old and new location or name), and the number of choices for the inequality constraints introduced by CA is limited to four (the number of different access modifiers; cf. Section 8.2).

In terms of the times required for generating (*gen*) and solving (*solv*) the CSPs representing the refactoring problems, Table 5 shows that across all filters and modes, *gen* (which includes querying a database representation of the program), is much larger than *solv*. In fact, while *solv* is negligible on average, *gen* can take up to 5 seconds, which is however still quite fast (but see below). This result reflects the fact that much of the effort previously burdened on the constraint solver has been shifted to the constraint generation phase, where it takes its toll.

Secondly, the time required for generating the constraints with the filters allowing a change of location (CL and CLAI) is significantly larger than for the rest. This is due to the fact that almost every constraint generated contains λ, λ_T, or π; if these are non-fixed,

Refacola's *GenerateConstraints* algorithm [18] will also look at all other properties contained in the constraint and, if variable, how they are constrained further.

The overall favourable times are relativized by three facts:

- The time for constraint generation does not include the time needed for filling the database against which the program queries are evaluated. We have excluded it since it depends largely on the infrastructure provided by the IDE in which *RefactoringWithForesight* is integrated (Eclipse in our case). The brute force approach that we used and that always analyses the whole program, regardless of the intended refactoring, can take up to 2 minutes for the larger projects of Table 3; although this can surely be optimized, one should bear in mind that many refactorings require a whole-program analysis.
- In all four modes, we did not submit generated constraints to the solver whose constrained properties all had fixed values: if such a constraint is satisfied, it does not contribute to the solution; if not, the whole CSP is not solvable. Note that "fixed values" here includes the properties of the refactoring intent, whose forced change to a new value must not be revised by the solver (even though in order to propagate the change, the properties count as non-fixed for *RefactoringWithForesight*; cf. the filter definitions in Table 2). This could of course have been handled by the constraint solver, but since all solvers we have experimented with had problems with large numbers of constraints, we added this optimization (whose integration in the algorithm of Figure 1 would have complicated its presentation).
- Times for generating constraints peaked at almost 2 minutes (for modes f^{++} and f^{+++} and filter CI) and for solving at slightly more than 30 seconds (for modes f^{+}, f^{++}, and f^{+++} and filter CLAI). This suggests that the threat to viability of constraint-based refactoring introduced by the foresight problem is very real; however, as evidenced by the relatively short average times that we observed, it can be counteracted in most cases.

9 Conclusion

Refactorings that change the structure of a program are subject to many restrictions, including ones that become apparent only after the structure has been changed. This is particularly a problem if not all structural changes are known in advance of the refactoring, for instance because some changes are dependent on others, or may or may not be needed to make a refactoring possible. To address this problem, we have identified two measures that complement each other. One turns parts of the precedents of constraint-generating rules to premises of conditional constraints, making the generated constraints more flexible in that they can adapt — during the constraint solution process — to structural changes of the program. The other is the introduction of quantified constraints representing an unlimited number of ordinary (non-quantified) constraints, constraining all conceivable changes that could be performed by a refactoring, including those that will actually be performed (which, therefore, need not be known in advance). Both measures have in common that they may generate significantly more constraints than actually needed for a specific refactoring; we have therefore devised an algorithm that keeps the number of additional constraints low. Experiments that we have conducted suggest that our algorithm can be highly effective, and that refactoring with foresight as proposed in this paper can indeed be feasible.

Acknowledgments. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant STE 906/4-1. The authors thank Andreas Thies for his contributions to the evaluation.

References

1. Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: Proc. of OOPSLA, pp. 265–279 (2005)
2. CHOCO Team choco: an Open Source Java Constraint Programming Library, Research Report 10-02-INFO, Ecole des Mines de Nantes (2010)
3. Donovan, A., Kiezun, A., Tschantz, M.S., Ernst, M.D.: Converting Java programs to use generic libraries. In: Proc. of OOPSLA, pp. 15–34 (2004)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
5. Fuhrer, R., Tip, F., Kieżun, A., Dolby, J., Keller, M.: Efficiently Refactoring Java Applications to Use Generic Libraries. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 71–96. Springer, Heidelberg (2005)
6. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, `http://java.sun.com/docs/books/jls/`
7. Gottlob, G., Greco, G., Mancini, T.: Conditional constraint satisfaction: logical foundations and complexity. In: Proc. of IJCAI, pp. 88–93 (2007)
8. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. PhD Dissertation, University of Washington (1992)
9. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in Java. In: Proc. of ICSE, pp. 431–440 (2008)
10. Kiezun, A., Ernst, M.D., Tip, F., Fuhrer, R.M.: Refactoring for parameterizing Java classes. In: Proc. of ICSE, pp. 437–446 (2007)
11. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proc. of AAAI, pp. 25–32 (1990)
12. Palsberg, J., Schwartzbach, M.I.: Object-Oriented Type Systems. Wiley (1994)
13. Schäfer, M., Ekman, T., de Moor, O.: Sound and extensible renaming for Java. In: Proc. of OOPSLA, pp. 277–294 (2008)
14. Schäfer, M., Dolby, J., Sridharan, M., Torlak, E., Tip, F.: Correct Refactoring of Concurrent Java Code. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 225–249. Springer, Heidelberg (2010)
15. Schäfer, M., de Moor, O.: Specifying and implementing refactorings. In: Proc. of OOPSLA, pp. 286–301 (2010)
16. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A comprehensive approach to naming and accessibility in refactoring Java programs. IEEE Trans. Soft. Eng. (2012)
17. Steimann, F., Thies, A.: From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009)
18. Steimann, F., Kollee, C., von Pilgrim, J.: A Refactoring Constraint Language and Its Application to Eiffel. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 255–280. Springer, Heidelberg (2011)
19. Tip, F., Kiezun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: Proc. of OOPSLA, pp. 13–26 (2003)
20. Tip, F., Fuhrer, R.M., Kiezun, A., Ernst, M.D., Balaban, I., De Sutter, B.: Refactoring using type constraints. ACM Trans. Program. Lang. Syst. 33(3), 9 (2011)

Magda: A New Language for Modularity*

Viviana Bono[1], Jarek Kuśmierek[2,3], and Mauro Mulatero[1]

[1] Dipartimento di Informatica, University of Torino, Italy
[2] Google Research, Krakow
[3] MIMUW, University of Warsaw, Poland

Abstract. We introduce Magda, a modularity-oriented programming language. The language features lightweight mixins as units of code reuse, modular initialization protocols, and a hygienic approach to identifiers. In particular, Magda's modularity guarantees that the client code of a library written in Magda will never break as a consequence of any addition of members to the library's mixins.

Keywords: modularity, mixin, constructor, accidental name clash.

1 Introduction

The limitations of mainstream object-oriented languages we are particularly concerned with, and which have been our motivations for the design of a new language, are the following.

- *Limitations of composition mechanisms.* The most widely applied composition and reuse mechanism is inheritance. In its classical, single-inheritance version, this mechanism is simple, however often not sufficient. In order to overcome its limits, other proposals were developed, like multiple inheritance, mixins [43,17,15,8,24,14,7] and traits [46,22]. Multiple inheritance is perceived as too complicated and too dangerous [20]. The two other constructs (mixins and traits) have been designed to tame multiple inheritance and managed to improve reusability in many aspects. However, they suffer from problems related to name clashes as described in Section 2.1.
- *Non-modular initialization protocol.* In most object-oriented languages, the initialization protocols are implemented as *constructors*. Constructors are responsible for the initialization of properties coming from different places in the class hierarchy, yet they are monolithic, which means that they have to perform all the job in one block of code. While a constructor can call a superclass' constructor to delegate part of the work, it still needs to keep all the superclass constructor's parameters in its own signature. This increases the amount of work which must be performed when the initialization protocol needs to be modified. Moreover, assumptions about the superclass' constructors are imposed, making those constructors more difficult to change.

* Work partially funded by the MIUR Project DISCO.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 560–588, 2012.

– *Accidental name clashes.* Classes can be implemented from different components (in standard languages, by extending other classes, or by implementing some interfaces); it may then occur that the same method name has different meanings in different components. This causes problems, from non-compilation to unexpected behavior during the program execution.

Our new language Magda is built around the notion of *mixin*. A mixin is a class parameterized over a superclass, introduced to model some forms of multiple inheritance and improve code modularization and reusability [43,17,15,8,24,14,7,3,45,16]. There are usually two operations defined on mixins: (*i*) application, by which a mixin is applied to a class to obtain a fully-fledged subclass (the class argument plays the role of the parent); (*ii*) composition, which makes a more specialized mixin by composing two existing ones. Notice that an indirect form of composition is possible even in the presence of application only, by applying a chain of mixins to a class (which is the way a *linearized multiple inheritance* is obtained). A mixin can defer definition and binding of methods until runtime, though attributes and instantiation parameters are still defined at compile time.

Our mixin construct has many points in common with its previous versions, with one noticeable difference: in Magda there is no concept of a class, therefore mixins are not interpreted as functions from classes to classes. Mixins are used directly to create new objects from, and also induce types in the nominal static type system of the language (as proposed independently in McJava [30]). Additionally, to take advantage of mixin reusability and enhance it, Magda contains two unique features. The most innovative one is the modularization of constructors, in such a way the part of the state initialization related to a feature is declared together with that feature. Then, mixins with independent initialization protocols can be combined without the need to copy any code. This feature was presented as a part of a Java extension called JavaMIP [13], and in [12] we proved the type soundness of Featherweight JavaMIP, an extension of Featherweight Java [29] with the modular initialization protocol. The second distinctive feature of Magda is the way of how declarations of new methods, overriding of existing methods, and method calls are specified, by declaring and referencing identifiers in a univocal way. This results in the absence of name clashes and accidental overridings. A version of this feature is the base of the HygJava language, presented in [37].

Magda was introduced in the second author's PhD thesis [35], where the language was presented together with its formal semantics and related properties. In this paper we present Magda by examples, with particular emphasis on the initialization design, which is the new and novel part of the language. In Section 2 we detail our motivations for introducing a new language with respect to the composition constructs and the initialization design, in Section 3 we present Magda by examples, in Section 4.3 we hint at the name-clashes related problems, in Section 4 we compare Magda to other languages, and Section 5 summarizes our work.

A proof-of-concept implementation of Magda, with examples (including an implementation of the Decorator pattern) and a how-to, is available [36].

Part of the material of this paper, notably the part present in Section 2 and in Section 4, was already presented in some form in [13], for motivating modular constructors for Java. Nevertheless, we believe it is necessary to include it also in the present paper, in order to motivate Magda's design choices.

2 Object-Oriented Languages: Limitations

In this section we present some limitations of the best known object-oriented languages, which gave us motivations to introduce Magda.

2.1 Limitations of Composition Mechanisms

The ultimate goal of object-oriented programming should be code reuse. However, this goal is still not reached completely, despite various constructs oriented to modularity introduced in different languages. We illustrate our point via a hierarchy:

- `BaseStream` - an abstract class (or interface), with abstract methods `read` and `write`.
- `FileStream` - a class representing streams which transfer data to and from a file.
- `NetworkStream` - a class representing streams transfering data across the network.
- `BufferedStream` - a class representing streams which buffer data before sending them in one big batch.
- `CompressedStream` - a class representing streams which compress and decompress data on the fly.
- `StatsStream` - a class representing streams which calculate different statistics.
- `EncryptedStream` - a class representing streams which encrypt data when written, and decrypt during reading.
- `DatabaseStream` - a class representing streams which transfer data to and from a database.

We assume that `BufferedStream` has a method `SetBufferSize`, which sets the amount of data in memory before the data is sent to the communication device. Moreover, class `CompressedStream` uses the method `SetBufferSize` to decide about the amount of data to be compressed. Additionally, we expect to be able to obtain combinations of the above features, like compressed and encrypted network stream, or buffered file stream with statistics.

Single Inheritance. With single inheritance, each class can have at most one ancestor. When we want a class to reuse features from more than one class, we

can make it inherit from the class which contains most of the needed features and then: either (*i*) we copy manually the code from the remaining classes; or (*ii*) we use object composition, declaring fields of the types of the left-out classes and then implementing methods associated to appropriate delegate objects. This approach can however cause problems, when all the classes in question share common ancestors defining a state. Then every change of the common state of the "proper object" needs to be propagated also to the delegate objects. This results in behavior which is in many aspects similar to the one of virtual multiple inheritance in C++.

Object Composition/Decorator Design Pattern. One of the approaches which exploit the object composition is the *Decorator pattern* [25]. In this approach, to create, for instance, an `EncryptedStream`, a new class is implemented which contains a reference to a stream object (called *delegate*). Then, in the new class we declare all the methods of the `BaseStream` class. Methods whose behavior is modified (like `read` and `write` in the stream hierarchy), perform their new tasks and then call the corresponding methods in the delegate. All other methods are implemented to just call their counterparts in the delegate object. This approach has often the required compositional flexibility, and it is sometimes preferred over inheritance. It is especially useful when additional features of objects should be enabled and disabled dynamically during the life of an object. However, it has numerous disadvantages when used instead of inheritance, because: it requires declarations of methods which will only call the same method in the delegate object; it creates unwanted dependencies in the code, because any modification or addition of a method in the delegate object's class requires a repetition of the same operation in other classes; when a class `A` is composed with a class `B` and redefines some of the methods from `B` by providing a new implementation in its declaration (that corresponds to method override in inheritance), then this redefinition is visible only from the point of view of external clients, since the other methods in the object calling this method will call the original implementation, not the redefined one.

Multiple Inheritance. Multiple inheritance, whose best known implementation is the one in C++, is a powerful feature, however its complicated semantics make it difficult and dangerous to use, as summarized by Cook [20]: "Multiple inheritance is good, but there is no good way to do it." The main problem with multiple inheritance is the way it deals with ambiguous and conflicting features. In the example, such problems would occur in a class inheriting from both `CompressedStream` and `BufferedStream`, because both classes have method `SetBufferSize`. Moveover, multiple inheritance does not allow the user to keep in the resulting class both of the methods with the same name. In addition, when the same method is defined or overridden in superclasses which share a common ancestor, then the order in which overridden variants of the method are called is not always obvious. As a result, many mainstream languages developed after

C++ (like Java and C♯), borrowed numerous features of that language, however not the multiple inheritance. There exist other languages which adopted different flavors of multiple inheritance, like Loglan [34] and Python [10]. Those languages use linearization algorithms to change the graph of ancestors into a list over which dynamic dispatch is performed. However, linearization leads to cases when the ordering of the overridings is non-trivial to understand and fragile to innocent-looking changes in the hierarchy.

Mixins. In the case of methods with the same name occurring in different mixins (like the aforementioned `SetBufferSize`), mixins permit explicit direct control of the order in which those methods will override each other, however, they might not allow the resulting class to have all variants. Nevertheless, there are two proposals offering this feature: MixGen [3] and MixedJava [24]. In the MixGen language, the class obtained from applying both `CompressedStream` and `BufferedStream` mixins will keep both variants, while giving access to one of them at each moment, depending on the static type of the variable referring to the object. However, this approach does not allow the user to access both of them at the same time and in some cases it is not obvious for the user which implementation will be called in a given expression.

Traits. Traits [46,22] have been designed as an alternative mechanism of code reuse. Initially they were introduced in a untyped setting as an extension of Smalltalk [22,46]. Later on they were studied also in typed (thus often restricted) settings [47,44,11,6]. One of the design goals of this approach was to overcome problems with mixins mentioned above. Each trait is a minimal unit of reuse, containing a set of implemented methods and a set of required methods. A class is then built by composing traits (with or without the presence of single inheritance). The advantage of the composition mechanism available in trait-based languages is that they issue warnings when name clashes occur and offers operators (like hiding, aliasing and renaming) to modify the way traits are composed in a class definition. Some additional code in the class, called *glue code*, might be needed to make the composition work.

In our example, one can specify traits `CompressedStream` and `EncryptedStream`, where each of them would require methods `read` and `write` and contain their overriding variants supporting compressed and encrypted data. Finally, using those traits, classes like `CompressedFileStream`, or `EncryptedNetworkStream` can be created. Thanks to this flexible method manipulation mechanism, the programmer has the choice of how to deal with the method names conflicts. However, this does not come without a price. Consider the following example (written in a Smalltalk-like syntax):

```
// Library containing trait EncryptedStream and class CertificateManager
trait named: #EncryptedStream
  instanceVariableNames: 'aCertificate'
  setCertificate: certificate
    ...do checks on the certificate...
  aCertificate := certificate.

  autoFindCertificate
    | manager |
    manager = getGlobalCertificateManager.
    manager findCertificateFor: self.

Object subclass: #CertificateManager
  findCertificateFor: encrypted_connection
    | some_certificate |
    some_certificate := ... do something to establish certificate....
    encrypted_connection setCertificate: some_certificate.


// Client code with class CryptFileStream and its usage
FileStream subclass: #CryptFileStream
  uses: {EncryptedStream @ {#setCertificate: -> #setEncryptCert:} -
         {#setCertificate:}} + otherTrait
     ... a class declaration ....

// Usage of CryptFileStream class.
// The call to method autoFindCertificate fails, because the renamed method
// setCertificate is used by the CertificateManager object (created internally).
// This dependency is not visible in the signature!!!!!!!!!!!!!!!!!!!!!!!!!
  stream := CryptFileStream new.
  stream autoFindCertificate.
```

Trait `EncryptedStream` encrypts data, and comes with a `CertificateManager` (trait or class). This `CertificateManager`, depending on the user logged in, sets certificates automatically for the encrypted streams. To do this, the `CertificateManager` calls method `setCertificate` on the `EncryptedStream` and also a getter to choose the certificate needed for the stream. Unfortunately, when the new `CryptFileStream` class will be defined using the `EncryptedStream` trait, and the `setCertificate` method will be renamed, then the certificate manager will stop working on that stream. Notice that such a dependency is not reflected in the specification of the methods required by a trait, because the latter lists only methods required to be present in the same object. In general, when traits are used to create functionalities spanning many objects, it might be hard to predict which dependency will be broken, that is, when any method in the trait will be renamed or hidden. This problem has already been spotted and worked around using different solutions. The first solution was developed in Chai, an extension of Java with traits [47], and independently in the Fortress language [6,4,5]. The second solution is in the work adapting traits to statically typed languages [44]. Finally, another solution is based on *freezable traits* [23]. In the first solution, when a method is removed, then another

implementation of the method with the same signature must be added to the class. However, those methods can sometimes be sematically different. Additionally, this restriction implies that conflicts of two different methods with the same name and different result type cannot be resolved. In the Fortress language, traits also induce types, therefore Fortress language is even more restrictive than Chai: method renaming and hiding are forbidden. As a consequence, when two traits with methods with the same name are composed, then one method overrides the other one. In the second solution, traits themselves are not visible in the nominal type system of the language. Therefore, it is necessary to declare a public interface containing (some of) the methods of a trait, and make sure that all classes using that trait implement the interface. This implies that, when a method is renamed in the declaration of a class, then it either requires the class to stop implementing that interface or imposes additional changes in the code. In the third solution based on the notion of method freezing [23], it is possible to keep two versions of conflicting methods: one as private, and other one as public. The private version is executed when a self call is performed, which means a call from the trait in which this private, or frozen, method has been declared. However, this approach can only be used to solve the problem when the conflicting method is called internally, by another method declared in the same trait. Unfortunately, it cannot be used when this method is required by another class/trait, as the method `setCertificate:`, called in method `findCertificateFor:` of class `CertificateManager`.

2.2 Non-modular Initialization Protocol

Class-based languages are usually equipped with an object initialization protocol. Such protocol describes: (*i*) the information needed to create and initialize an object; (*ii*) the code performing the initialization. In most languages (for instance, C++ [48], C♯ [28], Delphi [2], Java [27], and Visual Basic [1]), the initialization protocol of a class is specified by *constructors*. Each constructor consists of: (*i*) a list of parameters; (*ii*) a body. Unfortunately, this traditional approach to initialization (TIP from now on) has drawbacks. We present them via some Java examples, even though most of the problems occur also in other mainstream languages.

Optional Parameters. Traditional initialization protocol leads to an exponential number of constructors with respect to the number of optional parameters. In Java, in classes like `java.awt.TextArea`, there is often one constructor with the largest possible set of parameters. However, additional constructors with a subset of those parameters must be declared when some parameters are optional. Moreover, it is not possible to have two constructors with parameter lists of the same length and compatible types of the corresponding parameters in the same class. These drawbacks can be partially solved in languages containing named parameters (like Python [10]) and default parameter values (like Delphi [2], Python [10], C++ [48]).

Multiple Initialization Options and Code Duplication. TIP leads also to an exponential number of constructors with respect to the number of object properties with different initialization options. If each property can be initialized in more than one way, then the possible number of initialization options of a given class (thus the number of constructors) is a multiplication of the numbers of options of the object properties. An example could be the combination of a property color (with two palettes, RGB and CMYK) with a property position (with three options, cartesian, polar, and complex) in a class `ColorPoint`, which induces six constructors, with some duplication of code. The Java class `java.net.Socket`, which contains nine constructors, is a more sophisticated, real-life example of this problem.

Unnecessary Constructor Dependencies. It may happen that modifications of a class force unnecessary changes in subclasses. Consider the following scenario: class C_1 with a set of constructors; class C_2 declared as a subclass of C_1, containing all parent constructors redeclared by adding a parameter *Par'*, calling the corresponding constructors in C_1; if another constructor is added to C_1, C_2 will not inherit automatically the corresponding constructor. This class, depending on the language design, will either retain the constructor added in C_1 without the additional parameter *Par'*, or just will not inherit it at all (as in the case of Java). We think that neither of these options is good enough.

Fragile Overloaded Constructors. Overloaded constructors in TIP make safe-looking changes non conservative. When a Java (or C++ or C♯) class contains different constructors, the choice of the constructor is done in the same way as the choice of an overloaded method variant. Thus, it suffers the same problems, as this example shows:

```
interface I1 {...}
interface I2 {...}
class C1 implements I1, I2 {...}
class C2 implements I2 {...}

class ClassWithOptions
{   ClassWithOptions (I1 a, I2 b);
    ClassWithOptions (I2 a, I1 b);
}
 ...
new ClassWithOptions( new C1(), new C2() );
```

This code compiles, because only one of constructors of class `ClassWithOptions` matches the `new` expression. However, if the class `C2` implements also the interface `I1`, then the previous code will not work anymore (because the last `new` becomes ambiguous). Notice that this is not a problem in languages in which it is possible to name constructors (e.g., in Delphi [2]), since the overloading can be avoided.

Unnecessary Redeclarations of Checked Exceptions. In Java, if a constructor throws some exceptions and is called by a constructor of a subclass, this one must repeat the whole list of exception declarations. While such a repetition in

methods is important, because methods have choices (to catch the exceptions, or throw them further), the situation with constructors is different. A constructor cannot catch the exceptions thrown by a superclass' constructors, therefore the repetition of the declarations is an additional, useless work.

Problems with Traditional Mixins. A mixin declaration, like any other subclass declaration, may contain declarations of new constructors, which, in turn, may reference the superclass constructors. There are cases in which it would be desirable to compose independently designed mixins. We consider the following example: a class `Button` and two mixins `Blinking` and `Ringing`, that, when applied separately to `Button`, will result in, respectively, a class of blinking buttons and a class of ringing buttons. Then, we want to compose them to obtain a class of blinking and ringing buttons (the order should not matter). Unfortunately, if the mixins modify the interface of a constructor of the parent class, for example by adding one parameter, then they are non-composable. Assume that `Button` has a constructor with n parameters and that the mixin `Blinking` must define a new constructor (that replaces the old one), having n+1 parameters (one more for color) and calling the superclass constructor. Similarly, the `Ringing` mixin may need, for example, the frequency, therefore it will have an (n+1)-parameter constructor as well. Now, if we apply one of those mixins to the `Button` class, then the resulting class will have a constructor with $n + 1$ parameters, and it will not be an appropriate argument for the other mixin. Those problems have, in fact, a similar nature as those occurring with standard subclassing. However, mixins are designed for a wider reuse than subclasses, therefore problems may occur more often and are more difficult to foresee. Notice that also the designers of Jam [7] have encountered this problem. In order to simplify the matter, they decided to disallow the declarations of constructors in mixins, thus forcing programmers to write constructors manually in classes resulting from a mixin application.

3 The Magda Language

In this section we present the Magda language by examples. A detailed description of its syntax can be found in [35].

Every program in Magda consists of two parts: a list of *mixin declarations* and a list of instructions, called *main instructions*. The main instructions may use the mixin declarations. Any type in Magda is a sequence of mixin names. Values are either `null` or object references. Value `null` is the default value for variables and fields. Magda expressions include:

- the creation of a new object. Each object in Magda is created from a non-empty sequence of mixins. The sequence of mixins plays a role similar to the one of a class in other languages.
- the method call. In Magda, as in most object-oriented languages, the set of methods "understood" by an object depends on the "schema" from which it was created. Therefore, the set of understood methods depends on the sequence of mixins used to create the object.

This is a simple program printing "Hello World" written in Magda:

```
mixin HelloWorld of Object =
  new Object MainMatter()
  begin
    "Hello world".String.print();
  end;
end;
//
(new HelloWorld []).HelloWorld.MainMatter();
```

The program consists of the declaration of a mixin named `HelloWorld` and one main instruction. The mixin `HelloWorld` contains a method named `MainMatter`. This method begins with the keyword `new` which indicates the introduction of a *new method identifier*, as opposed to *method redefinition*, and *abstract methods*, both described later on. The main instruction starts with the creation of an object from mixin `HelloWorld`, using the `new HelloWorld[]` expression. Then it calls the method `MainMatter` on the object, that, in turn, calls the method `print`, declared in mixin `String`. In Magda, `String` is a built-in mixin and contains methods like `print` and `add` (which concatenates two strings). Similarly, Magda contains other built-in mixins like `Integer` and `Boolean`. The `Boolean` mixin cannot be used in object creation expressions. Then, the only way to obtain a `Boolean` value is to use one of the `Boolean` constants: `true` and `false`. Since booleans are object values, `null` is also the default value for `Boolean`. Moreover, Magda features two control instructions: a conditional instruction `if` and a loop instruction `while`.

In each method call, the method name is prefixed with the name of the mixin in which it was introduced. This is to enforce *hygiene* of identifiers in order to avoid accidental clashes, as explained earlier in the introduction.

In the current version of Magda all methods are visible. This results in a similar behavior as the one of `public` methods in Java. Visibility is however orthogonal to the Magda's specific features presented in this paper, therefore it is not discussed, but kept in mind for future versions of the language. In practice, for a better code organization, the declarations of mixins should be split into different modules, with their own namespaces. However, this issue could be dealt with as in other languages (like Java and *C*♯) and in this paper we assume that every name of mixin is unique.

Object Fields and Local Variables. All fields of an object are declared, with a name and a type, in the mixins from which the object is created. The type of a field is a sequence of mixin names. The value of each field can be either the `null` value or an object value, that is, a reference to an object. For simplicity, we have chosen that each field in a Magda object can be only accessed by methods of that object, via an invocation of the form `this.fieldId`. As a result we obtain a behavior close to the one of *protected* fields in other languages. Furthermore, similarly to method identifiers, all fields are prefixed with the name of the mixin in which the fields have been declared.

In the following example there is a declaration of a mixin `Point2D` containing two field declarations, `x` and `y`.

```
mixin Point2D of Object =
  x:Integer;
  y:Integer;

  new Object setCoords( ax:Integer; ay:Integer)
  begin
    this.Point2D.x := ax;
    this.Point2D.y := ay;
  end;

  new Integer getX()
  begin
    return this.Point2D.x;
  end;
end;
//
mixin MainClass of Object =
  new Object MainMatter()
    p1:Point2D;
    p2:Point2D;
  begin
    p1 := new Point2D[];
    p2 := p1;
    p1.Point2D.setCoords( 11, 10);
    p2.Point2D.getX().Integer.print();  //this line prints out 11
  end;
end;
//
(new MainClass []).MainClass.MainMatter();
```

As in `MainClass.MainMatter` method, variable declarations are in the header of the method, after the signature, before the keyword `begin`, (like in Pascal [9]). The type of a variable is a sequence of mixin names; in our example, variable `p1` has type `Point2D`. Similarly to fields, local variables can be assigned with object values.

Inheritance. The inheritance mechanism in Magda works in a different way than in traditional single-inheritance languages (like Java, C♯) or multiple-inheritance ones (like C++), where a new class extending an existing class or classes automatically includes all of their members (fields and methods).

In Magda, a mixin `C` can specify that it *requires* other mixins (with the keyword `of`), for instance mixin `A`. In this case, we say that `A` is a *base mixin* of `C`. Moreover, we call *indirect base mixins* the mixins beloging to the transitive closure of the relation "requires" with respect to the mixin declarations in a program. All code referring to an object created from mixin `C` can safely use methods declared in the base mixin `A`. This applies also to the code of the methods declared within mixin `C` itself. However, unlike in traditional languages, all the methods and the fields declared in `A` are not implicitly included in `C`, that is,

all the members declared within A are not visible in the same way as if they had been declared in C. In particular it means that: (*i*) an object creation expression using mixin C needs also to explicitly mention mixin A. Moreover, in the sequence of mixins used to create an object from, A must occur at an earlier position than C; (*ii*) every method call (and field dereference) on an object created from the set of mixins {A, C}, is prefixed with the name of the mixin from which the method originates. For instance, if the method was declared originally within A, then the method call needs to specify it. Each mixin which is not declared to extend explicitly any specific mixin extends implicitly Object which is the base mixin of every mixin.

A simple example of a program containing the declaration of two mixins, Point2D the base mixin and Point3D extending it, follows.

```
mixin Point2D of Object =
  x:Integer;
  y:Integer;

  new Object setCoords2D( ax:Integer; ay:Integer)
  begin
    this.Point2D.x := ax;
    this.Point2D.y := ay;
  end;

  new Integer getX()
  begin
    return this.Point2D.x;
  end;
end;

mixin Point3D of Point2D =
  z:Integer;

  new Object setCoords3D( ax:Integer; ay:Integer; az:Integer)
  begin
    this.Point2D.setCoords2D(ax, ay); //a call to the method coming from
    this.Point3D.z := az;             //another mixin is prefixed with its name
  end;
end;

mixin MainClass of Object =
  new Object MainMatter()
    x:Point3D;
  begin
    x := new Point2D, Point3D[];
    x.Point3D.setCoords3D( 10, 11, 12); //this method comes from Point3D
    x.Point2D.getX().Integer.print();   //while that one from    Point2D
  end;
end;
(new MainClass []).MainClass.MainMatter();
```

Multiple Inheritance. In this section we present how Magda mixins can be combined to obtain a form of multiple inheritance. The program in Figures 1 features the declarations of five mixins. The first two mixins `DisplayableObject`,

```
// Declarations of two mixins (DisplayableObject, Point2D)
mixin DisplayableObject of Object = ...
end;

mixin Point2D of Object =  ...
end;

// Two other mixins extending the Point2D mixin
mixin Point3D of Point2D =  ...
end;

mixin ColorPoint of Point2D = ...
end;

// A mixin extending three independent mixins
mixin Displayable3DColorPoint of DisplayableObject, Point3D, ColorPoint =  ...
end;

// Composition of independent mixins in object creation
new Point2D, Point3D, ColorPoint [...]
```

Fig. 1. Multiple inheritance in Magda: code snippets

`Point2D` do not inherit from any other mixin. Mixins `Point3D`, `ColorPoint` extend mixin `Point2D` in a single-inheritance fashion. Then, there is mixin `Displayable3DColorPoint`, extending three independent mixins (Figure 2 depicts the inheritance graph). Finally, the program contains the creation of an object, which uses three mixins to create an object from.

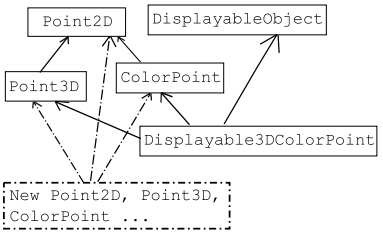


Fig. 2. Multiple inheritance in Magda: mixin hierarchy

In Magda there are two features for modularizing the code and composing components: (*i*) a mixin `A` can declare more than one base mixin, that is, it can extend multiple mixins at once; then, in an object creation expression using the mixin, all of its base mixins must be listed explicitly and the base mixins must occur earlier than `A` in the expression, however their relative ordering is arbitrary (see the example on virtual methods shown below, which presents different results depending on ordering); (*ii*) the declaration of base mixins represents only the minimal set of mixins required by a given mixin `A` to be combined with. Therefore, an object creation expression can combine more mixins with the mixin `A` than specified in its base mixin declarations.

Virtual Methods. Methods in Magda work like virtual methods of other languages (for example, C++ [48]). Unlike in the Java language, method redefinition has a different syntax from method introduction. The redefinition begins with the `override` keyword instead of the `new` keyword used in the introduction. Additionally, each method redefinition contains the name of the mixin in which the redefined method was first introduced. The body of a method redefinition can use a `super(...)` expression to call the previous implementation of the method. When a `super(...)` expression is evaluated in a method mt of a mixin M_c belonging to a mixin sequence $\overrightarrow{M}$, the previous implementation of mt is the one present in the last mixin in $\overrightarrow{M}$ preceding M_c, which contains a definition/redefinition of the method mt. As a result, a `super(...)` call expression within a method redefinition used in different objects can call different method implementations. That is, the actual method body called by the `super(...)` expression depends on the mixins (and their order) which have been used to create the object.

An example of a program using method redefinitions in which the behavior of `super(...)` call changes depending on the ordering of mixins is the following.

```
mixin BaseMixin of Object=
  new String GetActualName()
  begin
    return "Base ";
  end;
end;

mixin Extension1 of BaseMixin =
  override String BaseMixin.GetActualName()
  begin
    return super().String.add("Extension1 ");
  end;
end;

mixin Extension2 of BaseMixin =
  override String BaseMixin.GetActualName()
  begin
    return super().String.add("Extension2 ");
  end;
end;
```

```
(new BaseMixin, Extension1, Extension2[]).BaseMixin.GetActualName().String.print();
"\n".String.print();                    // this prints the "end of line" character
(new BaseMixin, Extension2, Extension1[]).BaseMixin.GetActualName().String.print();
// Program generates output of the form:
// Base Extension1 Extension2
// Base Extension2 Extension1
```

Because we choose explicitly which method from which mixin to redefine when declaring a method redefinition, there are no problems when two base mixins of a given mixin contain method definitions with the same name. Moreover, this explicitness also ensures that accidental name clashes are avoided when a new method is added to a mixin extended by other mixins.

Abstract Methods. A mixin declaration can also contain a declaration of a new method identifier without the body, marked with the keyword `abstract` (similarly to Java). Then, another mixin can supply its body, by marking it with the keyword `override`. If a mixin containing an `abstract` method is used in an object creation expression, then another mixin with the `override` version of the same method is required. The following example shows some features of the abstract methods.

```
mixin M1 of Object =
  abstract String Met1();
end;

mixin M2 of M1 =
  override String M1.Met1()
  begin
    return "Implementation from M2";
  end;
end;

mixin M3 of M1 =
  override String M1.Met1()
  begin
    return super().String.add(" with redefinition from M3");
  end;
end;

(new M1, M2, M3 []).M1.Met1().String.print();
// OK and prints: "Implementation from M2 with redefinition from M3"

(new M1, M3, M2 []);
// does not compile, M3.Met1 not suited as first definition of M1.Met1
// (contains call to super())

(new M1 []);
// does not compile, implementation of M1.Met1 missing
```

Object Initialization. We developed for Magda a *modular initialization protocol* approach based on small, composable pieces called *initialization modules* (*ini modules* from now on).

Each ini module has a list of formal parameters, called *input parameters*, which can be supplied at object creation. Its signature also declares whether its usage is *required* or *optional*, meaning whether if its input parameters are required or are optional in an object creation expression including the mixin containing that ini module. An ini module also specifies a list of *output parameters*, referring by name to input parameters declared in other modules, which will be computed by the ini module and supplied to such other modules.

The pseudo-syntax of an ini module is as follows:

```
<modifier> MixinName(in_param→) initializes (out_param_id→)
  local_var→
  I1
  super[out_param_id := expr→];
  I2
end;
```

where: `<modifier>` is either `required` or `optional`; `MixinName` is the name of the mixin in which the ini module is declared (we include it in order to make ini modules look similar to constructors in Java and C♯); $\overrightarrow{\texttt{in\_param}}$ is the (possibly empty) list of the input parameters of the ini module, declared with their types; $\overrightarrow{\texttt{out\_param\_id}}$ is the (possibly empty) list of the output parameter (hygienic) identifiers whose values are computed by the ini module; $\texttt{super[}\overrightarrow{\texttt{out\_param\_id := expr}}\texttt{]}$ is an assigment to the output parameters and a call to other ini modules, which the output parameters will be supplied to as input parameters.

The pseudo-syntax of an object creation instruction providing initialization parameters is as follows:

```
obj_id := new mixin_sequence[id := expr→];
```

where: `mixin_sequence` is the sequence of mixins the object is created from; $\overrightarrow{\texttt{id := expr}}$ are the initialization parameters together with their initialization expressions.

We describe the semantics of the object initialization procedure with an informal algorithm (the complete formalization of the semantics can be found in [35]).

Given an object creation expression `new mixin_sequence[`$\overrightarrow{\texttt{id := expr}}$`]`:

- the $\overrightarrow{\texttt{expr}}$ are evaluated in $\overrightarrow{\texttt{val}}$ and assigned to their each respective `id`;
- (*) a sequence of ini modules $\overrightarrow{\texttt{mod}}$ is extracted from `mixin_sequence`, respecting the order of the mixins in `<mixin_sequence>` and, within each mixin, respecting the syntactic order from top to bottom (i.e., a module declared first in the mixin comes first in the sequence). An ini module is a tuple containing: the sequence of input parameters, the sequence of output parameters, the body, the name of the mixin it comes from.

Then, consider a set of assigned parameters $\overline{\texttt{id} := \texttt{val}}$ and a sequence of ini modules $\overrightarrow{\texttt{mod}}$:

- if the set of assigned parameters and the sequence of ini modules are empty, then the initialization process terminates;
- otherwise, if the sequence of ini modules is not empty, let `mod` be the last ini module of the sequence, `Ip` be the set of input parameter identifiers of `mod`, and the instruction `super[`$\overrightarrow{\texttt{out\_param\_id := expr}}$`]` be contained in the body of `mod`:
  - if `Ip` is not equal to any subset of $\overline{\texttt{id}}$ and `Ip`$\neq \emptyset$, then the inizialization process is resumed with the sequence of ini modules $\overrightarrow{\texttt{mod}}$`/mod` (`mod` is discarded) and the same set of assigned parameters $\overline{\texttt{id} := \texttt{val}}$;
  - if `Ip` is a subset of $\overline{\texttt{id}}$ we say that `mod` is *activated* and we proceed as follows: (*1*) the identifiers in `Ip` are mapped onto their actual values (which are among the $\overline{\texttt{val}}$ in $\overline{\texttt{id} := \texttt{val}}$); (*2*) the sequence of instructions `I1` is executed; (*3*) the expressions in `super[`$\overrightarrow{\texttt{out\_param\_id := expr}}$`]` are evaluated and assigned to their respective output parameter identifiers; (*4*) the initialization process is resumed with the sequence of ini modules $\overrightarrow{\texttt{mod}}$`/mod`, and the set of assigned parameters where the ones corresponding to the identifiers in `Ip` are removed and the assigned output parameters of `mod` are added; (*5*) the sequence of instructions `I2` is executed.

A static check ensures that all `required` ini modules are activated (see [35]). Note that ini modules with an empty sequence of input parameters are always activated.

We say that a parameter `par` is *not consumed* if it is either supplied in an object creation expression, or calculated as an output parameter of an already executed ini module, but no module taking `par` as an input parameter has been executed yet; *consumed*, otherwise.

It is important to remark that the order in which ini modules are written in a mixin has an impact on the object creation. In particular, considering the order in which the sequence of ini modules is built (see the step marked with (*)) and then examined during the initialization process, if an ini module `mod1` in a mixin outputs a parameter which is consumed by another module `mod2` in the same mixin, then `mod2` must be written above `mod1`. This condition is verified by the static checker (see [35]). The need of a syntactic ordering among ini modules in a mixin may look inconvenient, but it seems difficult to discharge it without complicating the operational semantics of the modular initialization protocol in a significant way. In [26], there is a version of JavaMIP [13] where the ordering among ini modules in a mixin can be random; this version has a simple semantics, but some restrictions on the modular protocol are present. For example, it is not possible to split the initialization of a particular version of a property over more than one ini module (for instance, there would be no possibility of initializing `x` and `y` of the example in Figure 3 with two different ini modules, one for each).

We present a program based on two mixins containing ini module declarations in Figure 3. The corresponding executable code can be downloaded [36].

The first mixin (`Point2D`) contains the declaration of ini module `mod1`, requiring parameters `x` and `y` and not computing any output parameter. That is why the `super[]` instruction in `mod1` does not contain any parameter assignment.

The second mixin (`Point3D`) contains the declarations of two ini modules. The first one (`mod2`) expects one input parameter `z`, which is used to initialize the field `fz` of the object. The second one (`mod3`) has one input parameter `other` and three output parameters. Those output parameters refer to the input parameter of the module `mod2` declared in the same mixin, and to the parameters of the module `mod1` declared in the base mixin. Its body contains a `super[...]` instruction, which computes the values of the three output parameters. Finally, the `MainClass.MainMatter` method contains two object creation expressions. The first expression supplies the values of three initialization parameters: `z` of mixin `Point3D`, and `x` and `y` of mixin `Point2D`. The first parameter `Point3D.z` is supplied to module `mod2`, and that module is the first one to be executed. When its execution reaches `super[]`, the search begins for the next module to be executed, which is `mod1`, because parameters `x` and `y` are yet to be consumed. The last instruction of the module `mod1`, the `super[]` call, does nothing, since all parameters have been consumed. The second object creation expression supplies the value of one initialization parameter, `Point3D.other`, to the optional module `mod3`. When this module is executed, it computes the three parameters `x`,`y`,`z`. The third parameter is passed to the module `mod2` and then, when the execution reaches the `super[...]` call, the remaining two parameters are supplied to module `mod1`. As a result, all the ini modules declared in the program are executed during the creation of that object, in a bottom-up order: `mod3`, `mod2`, `mod1`.

Notice that the `super[...]` instruction in an ini module is different from the `super(...)` expression within overriding method bodies, because the parameters supplied in `super[...]` do not have to be the parameters which will be passed to the ini module activated by this `super[...]` instruction. Those parameters can be in fact passed to other modules which will be executed later on. To understand this better, consider again the example in Figure 3. The `super[...]` instruction in the module `mod3` supplies the values of three initialization parameters and calls the module `mod2`. However, only one of these parameters (`z`) is consumed by `mod2`, while the remaining parameters (`x`, `y`) will be supplied to another ini module (`mod1`). Moreover, `super[ ]` instruction within `mod2` does not contain any parameters, however it calls the module `mod1` which takes two parameters which have been computed by `mod3`.

In the following, we will discuss how Magda's ini modules solve the problems of traditional constructors described in Section 2.

Optional Parameters. We can introduce an optional ini module for each set of optional attributes as (input) parameters without increasing exponentially the number of ini modules with respect to the number of the attributes, since each ini module takes care only of its input parameters. Also, there is no problem

```
mixin Point2D of Object =
  fx:Integer; fy:Integer;

  required Point2D(x:Integer; y:Integer) initializes ()         //module mod1
  begin
    this.Point2D.fx := x;
    this.Point2D.fy := y;
    super[];
  end;

  new Integer getX() ....;
  new Integer getY() ....;
end;

mixin Point3D of Point2D =
  fz:Integer;

  required Point3D(z:Integer) initializes ()                    //module mod2
  begin
    this.Point3D.fz := z;
    super[];
  end;

  optional Point3D(other:Point3D) initializes
      (Point2D.x, Point2D.y, Point3D.z)                         //module mod3
  begin
    super[Point2D.x:= other.Point2D.getX(), Point2D.y:= other.Point2D.getY(),
          Point3D.z:= other.Point3D.getZ()];
  end;

  new Integer getZ() ....;
end;

mixin MainClass of Object =
  new Object MainMatter()
    p1:Point3D; p2:Point2D;
  begin
    p1:= new Point2D, Point3D[ Point3D.z:= 12, Point2D.x:=10, Point2D.y:=11 ];
    p2:= new Point2D, Point3D[ Point3D.other := p1 ];
  end;
end;
(new MainClass []).MainClass.MainMatter();
```

Fig. 3. Object initialization in Magda

with parameter lists of the same length and compatible types, thanks to named parameters.

Unnecessary Constructor Dependencies. Any new ini module introduced in a mixin is independent from the ini modules already present. We see this in the following example (continuation of the example from Figure 3):

```
mixin ColorPoint of Point2D =
  fcr:Integer;
  fcg:Integer;
  fcb:Integer;

  required ColorPoint(cr:Integer, cg:Integer, cb:Integer) initializes ()
  begin
    this.ColorPoint.fcr := cr;
    this.ColorPoint.fcg := cg;
    this.ColorPoint.fcb := cb;
    super[];
  end;
end;

new Point2D, Point3D, ColorPoint[Point2D.x := 1, Point2D.y := 2,
                  Point3D.z := 10, ColorPoint.cr := 255, ....];
```

The initialization module in mixin `ColorPoint` is independent from the ones in mixin `Point2D`, in particular it does not refer to any of `Point2D` ini modules' parameters. An ini module contains references to input parameters of other ini modules only when it computes their actual values (in this case, they are present as output parameters of the ini module); for instance, the ini module from Figure 3 with input parameter `other:Point3D` refers to parameters `x`, `y`, and `z` as its output parameters.

Multiple Initialization Options and Code Duplication. Each option corresponds to a certain ini module, and each ini module deals with one option without need of code duplication. An example is the one above that mixes color with coordinates: this is a typical case that would cause code duplication when using classical constructors.

Fragile Overloaded Constructors. In Magda, the choice of the appropriate ini modules is done using the names of the parameters and no form of overloading is present. As a result, there can be no ambiguities.

Unnecessary Redeclaration of Checked Exceptions. A natural way of adding checked exceptions to Magda would be to add them to ini modules. Then, there would be no need to repeat the declarations in the ini modules of the derived mixins, since they work as extensions, not as replacements of the parent initialization modules.

Problems with Traditional Mixins. Since within mixin-based inheritance initialization problems are amplified with respect to the ones in class-based inheritance, we believe that ini modules are a natural way to enhance mixin modularity.

Types and Type Expressions. Magda is a statically typed language and enjoys a type soundness property guaranteeing absence of message-not-understood errors. This is formally defined and proved in [35]. What distinguishes Magda from other languages, then, is the way the type expressions are formed and the way the subtyping is verified. Every type in a Magda program is a sequence of mixin names. The ordering of mixin names in a type expression is insignificant. When a variable or a field is declared of type T, it means that its value can be either `null` or an object created from a sequence of mixins which contains at least the mixins present in T (even though it can contain more), except the mixin `Object` which is always used implicitly during each object creation. Similarly, if a method parameter is of type T, it means that in each method call the actual value must be an object created from all the mixins in T (and maybe more).

For example, consider the program in Figure 3. The type of variable `p1` in method `MainClass.MainMatter` is `Point3D`, while the type of variable `p2` is `Point2D`. The second declaration means that the variable `p2` can hold only `null` value, or a reference to an object created using a sequence of mixins containing `Point2D`. However, notice that `Point3D` mixin has `Point2D` as its base mixins. As a result, every object created from `Point3D` is also created from `Point2D`. Therefore the type `Point2D, Point3D` is equivalent to the type `Point3D` as well as to the type `Point3D, Point2D`. On the one hand, in an object creation expression the base mixins cannot be omitted because the order in which they are present in the object creation expression is significant (see again the example on virtual methods). On the other hand, in types the ordering and the removal of base mixins are insignificant.

We say that type T_2 is the *fully expanded form* of type T_1 if T_2 is the type obtained by appending to T_1 all direct and indirect base mixins of T_1. In the example in Figures 1 the fully expanded form of type `Displayable3DColorPoint` is: `Object`, `Point2D`, `DisplayableObject`, `Point3D`, `ColorPoint`, `Displayable3DColorPoint`.

Now consider the program from Figure 1, extended with the mixin:

```
mixin Test of Object =
  new Object SomeMethod ()
    v1: Point3D;
    v2: Point3D, ColorPoint;
  begin
    ...
    v1 := v2; //OK
    v2 := v1; //not OK
  end;
end
```

In the method `SomeMethod` there are two variables, `v1` and `v2`. The requirements enforced by the type of variable `v2` are stricter than the ones enforced by the type of variable `v1`. As a result, each value of variable `v2` can be also a value of variable `v1`. However, the opposite does not hold. Therefore, the first assignment present in this method is type correct, however the second one is not. Thus this program will not compile.

In general, we say that type T_2 is a *subtype* of type T_1 (denoted $T_2 \preceq T_1$) when the fully expanded form of T_1 is a subset of the fully expanded form of T_2. As a consequence of this definition, we define the type of `null` value as the set of all mixin names used within a program.

Properties of Magda. Our modular approach to initialization is motivated by the fact that we want each mixin to be as composable with other mixins as possible, in order to minimize the amount of the code to be written (or, worse, duplicated). Magda enjoys an unusual safety property which intuitively means that no accidental conflicts can happen.

We say that an identifier `p` is a *transitive output parameter* of an ini module `m` if: (*i*) either `p` is an output parameter of `m`; (*ii*) or `p` is an output parameter of an ini module `n`, such that at least one input parameter of `n` is a transitive output parameter of `m`.

We say that two mixins `M_1` and `M_2` are *exclusive* if: (*i*) there exists a mixin `M_b` which is a direct or an indirect base mixin of both `M_1` and `M_2`; (*ii*) and there exists an input parameter `ip_b` of an ini module in `M_b`, such that both `M_1` and `M_2` contain a required ini module each, `m_1` and `m_2`, and `ip_b` is a transitive output parameter of `m_1` and `m_2`.

We say that a sequence of mixins $\overrightarrow{M}$ is *valid* if `new` $\overrightarrow{M}$ `[`$\overrightarrow{\texttt{par} := \texttt{expr}}$`]` is typable in the type system in [35], for some $\overrightarrow{\texttt{par} := \texttt{expr}}$.

We can now sketch Magda's *safety property*: for every two valid sequences of mixins $\overrightarrow{M_1}$, $\overrightarrow{M_2}$ where each pair of mixins taken one from $\overrightarrow{M_1}$ and one from $\overrightarrow{M_2}$ are not exclusive, any sequence of mixins obtained by combining $\overrightarrow{M_1}$, $\overrightarrow{M_2}$, in such a way the original reciprocal order of the mixins in each sequence is retained, is also a valid sequence.

Intuitively, this property ensures that if there is no ambiguity in the choice of ini modules (non-exclusivity), then any sequence of mixins can be combined with another.

Magda enjoys another important property: as hinted before, Magda's modularity guarantees that client code of a library written in Magda will never break as a consequence of any addition of members to the library's mixins (modulo exclusivity).

The property is as follows: for any set of well-typed mixins and any well-typed client code which uses it, if it is possible to extend a mixin in the set with a new method, or to add a new optional ini module to it, in such a way that the mixin itself is still well typed, Magda guarantees that: (*i*) all the other mixins in the set and the client code will still be well typed (i.e., they will still compile); (*ii*) the result of the execution of the client code will not change.

This property does not extend to the case of method override, however this is unavoidable, as override may change the semantics of a method. Magda's features are such that this is the only case in which it happens, as opposed in other languages, where also additions of members can cause unexpected changes.

Both properties can be proved by induction on the type derivation (and on the operational semantics execution tree), as defined in [35].

4 Related Work

In this section, we compare our solution with other approaches to reusability of components and we discuss different views to modularization of the initialization protocol. Moreover, we discuss how encapsulation works in Magda.

4.1 Code Reuse Mechanisms

Multiple Inheritance. The most popular implementation of the multiple inheritance is present in C++ [48]. Other versions are present also in Dylan [21], Python [10], and Loglan [34]. However, all of them suffers from the problems discussed in Section 2.1. One of the features of C++ multiple inheritance is private inheritance. This feature also influences multiple inheritance: when a class A inherits from two classes B_1 and B_2, where each of those inherit privately from a class C, then class A will in fact contain two instances of class C which are not visible via the public interface. One of those instances will be visible by the B_1 part of A, while the other one through the B_2 part of A. Private inheritance is not directly available in Magda, however, this could be simulated via object composition.

Traditional Mixins. It is believed that one of the main reasons why mixins have not yet achieved a wide acceptance is the *fragile class hierarchies* problem [42], which is avoided in Magda, thanks to our hygienic approach to identifiers. Another difference between Magda and most of the other mixin-based proposals is that in the latter the mixin is a construct which is used to transform one class into another, and classes as well as interfaces still play a significant role in such languages. Instead, in Magda the mixin is the only inheritance construct and it is exploited to create objects from, to reuse code, and to define nominal types. As a consequence, the requirements on the "parametric superclass" in Magda (that is, the base mixin) are also specified using a sequence of mixin names.

CZ. The CZ proposal [40] addresses the diamond problem directly, without restricting the expressive power of multiple inheritance, by eliminating multiple inheritance implementation hierarchies in favour of multiple inheritance subtyping hierarchies. However, this approach forces the programmer to have almost twice as many classes as within a traditional hierarchy, moreover it does not solve the problems about the modularity of constructors.

4.2 Modularization of Constructors

Avoiding the Initialization Protocol. A class can be written using the approach of avoiding explicit initialization protocols by inserting one parameterless constructor (or none), while the real initialization process is implemented in ordinary methods. Unfortunately, with this approach, there is no direct way to check if the object is properly initialized. The programmer may create an object from the class and: (*i*) forget to call some of the methods responsible for the initialization; (*ii*) call too many of them; (*iii*) call them in an incorrect order; without

being warned by the compiler. Dynamic checks can be added by using formal specification languages, like JML [38] and the Design by Contract in Eiffel [41], but the assertion can grow complicated, moreover static checks would be more desirable.

Container Parameter. Constructors may have one parameter of a "container" type, such as `Vector` or `Dictionary`. The container contains values of all the initialization parameters, for instance indexed by their names. Then, the constructor performs a dynamic verification (a static one is impossible), that can become complicated if there are many initialization options.

Container Classes. The idea behind the *container classes* design pattern is to use a class for passing the set of parameters used to initialize a property. Such a class must have as many constructors (with their respective parameters) as the possible options of initialization of the given property. The use of such an approach allows one to avoid the problems of: (*i*) exponential growth of the number of constructors; (*ii*) unnecessary code duplication. However, it is necessary to preview the need of the pattern even if in earlier versions of a class there is only one option of initialization for each property. Moreover, it works only in cases when the set of options of initialization of a class coincides with the cartesian product of the initialization options of its properties. It cannot be used in more complicated cases, for example when there is an option of initialization using one value to initialize more than one property (using a container classes each), or when there is the need to initialize a subset of fields, all packaged into one contaneir class.

Optional Parameters. Constructors with *default* parameter values (present, for example, in Delphi [2] and C++ [48]) make it possible to declare fewer constructors, being able to treat some parameters as optional. This mechanism does not help, though, when it is necessary to have a mutually exclusive choice among different parameters (of different types), because one cannot limit which combinations of optional parameters are allowed. This solution works well used together with referencing parameters by their names.

Referencing by Name. Another useful feature which can be found in some languages (but not in any of the main-stream ones like Java and C♯) is referencing parameters of methods and constructors by their names. Such approach, present for example, in Flavors [43], Objective-C [33], and Ocaml [39], solves two problems: (*i*) it discards some ambiguities (caused by constructor overloading), because parameters with the same (or compatible) type can have different names; (*ii*) it allows a wider use of default parameter values. However, this feature only solves problems of optional parameters and discards some ambiguities, but does not prevent an exponential number of constructors and code duplication in the case of multiple options of initialization of orthogonal object properties.

Constructor Propagation in Java Layers. The Java Layers language [18] has a feature called *constructor propagation*, which can be illustrated by an example[1].

```
class Class1
{ propagate Class1(String s)  {I_1;}
  propagate Class1(int i)     {I_2;}
}
class Class2<T> extends T
{ propagate Class2(double j)  {I_3;}
  propagate Class2(boolean k) {I_4;}
}
```

Class `Class2<Class1>` has all the combinations of the "propagated constructors" of both classes, which means containing the same list of constructors as the following class.

```
class Class3
{ Class3 (String s, double j) {I_1; I_3;}
  Class3 (int i   , double j) {I_2; I_3;}
  Class3 (String s, boolean k) {I_1; I_4;}
  Class3 (int i   , boolean k) {I_2; I_4;}
}
```

This approach solves the problem of the exponential number of constructors if the sets of options of parameters are in different classes. However, this can only produce a cartesian product of sets of constructors. In fact, if we want to implement a subclass of a class `C` with the purpose of adding a new option for initializing a property declared in `C`, then we cannot do this using Java Layers. To better understand this, we consider the following example written in Magda, which is an extension with a new palette of the example of the colored cartesian point discussed in Section 2.2, *Multiple initialization options and code duplication.*

```
mixin HSBColorPoint of ColorPoint =
 optional HSBColorPoint(h:float, s:float, b:float)
   initializes (ColorPoint.r, ColorPoint.g, ColorPoint.b)
 begin
   ...
 end;
end;
```

This code, in the Java Layers approach, would require copying all the combinations of the propagated constructors.

Object Factories. A recent work concerning initialization protocols is [19]. It shows how object factories can be integrated in a language like Java in such a way that they use the same syntax as normal object creation. Using this approach,

[1] This is a modified version of an example taken from the web page of the Java Layers `http://www.cs.utexas.edu/~richcar/cardoneDefense.ppt`. The syntax is also slightly modified to look more Java-like.

it is possible to override the constructors of a class, and even to write an object creation expression of the form `new` I`(...)`, where I is an interface, thus giving more flexibility to the initialization code. As an effect, in some situations this reduces the amount of code which needs to be written. For example, when we want to add one initialization option to an existing class, we just extend the list of the constructors of that class. The benefits of this approach are: (*i*) the separation of the initialization from the class itself, so that a client instantiating an interface can even not know the implementing class; (*ii*) the possibility of modifying an initialization protocol in a way in which, for instance, the object returned by `new` expression is an already existing one (not a newly created one). However, in this approach, each set of initialization parameters must correspond to one constructor, therefore, in general, it does not avoid the problem of the exponential growth.

4.3 Dealing with Accidental Name Clashes

The concept of method is realized by three different actions: (*i*) the introduction of a new method; (*ii*) the implementation/override of an existing method; (*iii*) the method call. The bindings between (*ii*) and (*i*), as well as between (*iii*) and (*i*) are typically made using the method name, which is not guaranteed to be univocal. Additionally, in many popular languages (like Java, C♯ and C++), the distinction between (*i*) and (*ii*) is also based upon names. Therefore, modifications of existing classes (even conservative ones) may introduce errors.This can occur even more frequently, and it is more difficult to predict, when the modifications happens in a library written by third-party developers.

In the work [37] we presented three kinds of ambiguity problems (name clashes caused by the implementation of an interface, name clashes caused by the addition of a new method, name clashes caused by mixin application), together with an empirical analysis of the Java standard library (1.5) from the point of view of name clashes.

There are various proposals in the literature to tackle accidental name clashes. Notably, there are languages offering constructs and mechanisms to deal with conflicts: Delphi [2], C♯ [28], Eiffel [41], MixedJava [24], MixGen [3],[2], McJava [32,31] are some examples. An analysis of the main features of these proposals can be found in [37].

4.4 Encapsulation in Magda

Another characteristic which makes our approach different from most of other approaches to component reuse is encapsulation. In Magda, the set of mixins from which an object is created is always visible, since all the references to methods and fields need to be prefixed with the name of the mixin from which the method or field comes. This is in contrast with the choices made in most of the other languages, and at the first sight it may look as a drawback. However,

[2] They introduced the notion of *hygienic mixin*.

it is often the case that the internal structure of a class is not completely transparent to the user, even though a form of encapsulation is enforced. In C++, for example, in the case of multiple inheritance it is necessary to know if there are common superclassess, in oder to choose one of the semantics of inheritance (private, public or virtual). In the cases of MixGen [3] and MixedJava [24], a class can have many distinct implementations of one method. Then, to call a method declared in a specific superclass or mixin, one has to cast the type of an object to that specific type. In the case of freezable traits [23], to unfreeze a method in a trait or class one needs to know in which supertrait this method has been declared. Additionally, the user needs to be aware of which methods are called in which other methods (as described in Section 2.1), therefore the encapsulation is also violated even at the level of methods.

5 Conclusions

Our purely mixin-based design offers: (*i*) a simple mechanism for reuse based on mixin inheritance, without the problems present in other implementations of the mixin construct (see Section 2.1); (*ii*) initialization protocols that can be composed from independent ini modules coming from different mixins, thus avoiding the drawbacks described in Section 2.2; (*iii*) a mechanism for identifier references using fully qualified names, which guarantees that programs will never fail to compile, or behave unexpectedly, as a result of changes causing name clashes (this way we avoid the problems hinted in Section 4.3). The reuse mechanism, together with the modular constructors and the hygienic identifiers, provides modularity: it is guaranteed that the client code of a library written in Magda will never break as a consequence of any addition of members to the library's mixins.

The hygienic approach to identifiers improve inter-component compatibility. It can be argued, though, that full-path references are lengthy to write and difficult to read, and that this is one main drawback of Magda. However, this can be solved by implementing an IDE tool for expanding short names and hiding long ones on demand. The tool could be based upon the following rule: for each non-hygienic method or field identifier in an expression, find the first mixin in the type of the expression such that it contains the introduction of an identifier with the same name, where introduction means the first declaration of the identifier in the mixin hierarchy. In case of ambiguity, all possible valid hygienic hierarchies would be shown.

Acknowledgments. The authors would like to thank the anonymous referees.

References

1. Visual Basic .NET Language Reference. Microsoft Press (2002)
2. Delphi Language Guide. Borland Software Corporation (2004)

3. Allen, E., Bannet, J., Cartwright, R.: A first-class approach to genericity. In: Proc. OOPSLA 2003, pp. 96–114 (2003)
4. Allen, E., Chase, D., Flood, C., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L.: Project Fortress: A multicore language for multicore processors. Linux Magazine, 38–43 (September 2007)
5. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress language specification. Technical report, Sun Microsystems (2008)
6. Allen, E., Hallett, J.J., Luchangco, V., Ryu, S., Steele Jr., G.L.: Modular multiple dispatch with multiple inheritance. In: Proc. SAC 2007, pp.1117–1121. ACM (2007)
7. Ancona, D., Lagorio, G., Zucca, E.: Jam - A Smooth Extension of Java with Mixins. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 154–178. Springer, Heidelberg (2000)
8. Ancona, D., Zucca, E.: An Algebra of Mixin Modules. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 92–106. Springer, Heidelberg (1998)
9. Barron, D.W. (ed.): Pascal: The Language and its Implementation. John Wiley (1981)
10. Beazley, D., Van Rossum, G.: Python. Essential Reference. New Riders Publishing, Thousand Oaks (1999)
11. Bono, V., Damiani, F., Giachino, E.: On Traits and Types in a Java-like Setting. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Proc. IFIP-TCS 2008. IFIP AICT, vol. 273, pp. 367–382. Springer, Boston (2008)
12. Bono, V., Kuśmierek, J.D.M.: FJMIP: A Calculus for a Modular Object Initialization. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 100–112. Springer, Heidelberg (2007)
13. Bono, V., Kuśmierek, J.D.M.: Modularizing constructors. Journal of Object Technology, Special Issue: TOOLS EUROPE 6(9), 297–317 (2007)
14. Bono, V., Patel, A., Shmatikov, V.: A Core Calculus of Classes and Mixins. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 43–66. Springer, Heidelberg (1999)
15. Bracha, G.: The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD thesis, The University of Utah (1992)
16. Bracha, G., Ahe, P., Bykov, V., Kashai, Y., Mirand, E.: The Newspeak programming platform. Technical report, Cadence Design Systems (2008)
17. Bracha, G., Cook, W.: Mixin-based Inheritance. In: Proc. OOPSLA 1990, pp. 303–311. ACM Press (1990)
18. Cardone, R.J.: Language and Compiler Support for Mixin Programming. PhD thesis, The University of Texas at Austin (2002)
19. Cohen, T., Gil, J.: Better construction with factories. Journal of Object Technology 6(6), 109–129 (2007)
20. Cook, S.: OOPSLA 1987 panel P2 varietes on inheritance. In: OOPSLA 1987 Addendum to Proc., pp. 35–40. ACM Press (1987)
21. Craig, I.D.: Programming in Dylan. Springer-Verlag New York, Inc., NJ (1996)
22. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.: Traits: A mechanism for fine-grained reuse. TOPLAS 28(2), 331–388 (2006)
23. Ducasse, S., Wuyts, R., Bergel, A., Nierstrasz, O.: User-changeable visibility: resolving unanticipated name clashes in traits. In: Proc. OOPSLA 2007, pp. 171–190. ACM (2007)
24. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and Mixins. In: Proc. POPL 1998, pp. 171–183. ACM (1998)

25. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)
26. Gialluca, M.: Modular initialization protocol: a new implementation of the JavaMIP language. Tesi di Laurea Trieannale, Torino University, Dipartimento di Informatica (2010)
27. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification. Addison-Wesley, Sun Microsystems (2005)
28. Hejlsberg, A., Golde, P., Wiltamuth, S.: C♯ Language Specification. Addison-Wesley (2003)
29. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. TOPLAS 23(3), 396–450 (2001)
30. Kamina, T., Tamai, T.: McJava – A Design and Implementation of Java with Mixin-Types. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 398–414. Springer, Heidelberg (2004)
31. Kamina, T., Tamai, T.: Flexible method combination based on mixin subtyping. Journal of Object Technology 4(10), 95–115 (2005)
32. Kamina, T., Tamai, T.: Selective method combination in mixin-based composition. In: Proc. SAC 2005, pp. 1269–1273 (2005)
33. Kochan, S.: Programming in Objective-C. Sams (2004)
34. Kreczmar, A., Salwicki, A., Warpechowski, M.: LOGLAN 1988—report on the programming language. Springer (1990)
35. Kuśmierek, J.: A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming. PhD thesis, Warsaw University, Departement of Informatics (2010), http://www.mimuw.edu.pl/~jdk/mixiny.pdf
36. Kuśmierek, J., Mulatero, M.: The Magda language implementation, http://sourceforge.net/projects/magdalanguage
37. Kuśmierek, J.D.M., Bono, V.: Hygienic methods, Introducing HygJava. Journal of Object Technology, Special Issue: TOOLS EUROPE 6(9), 209–229 (2007)
38. Leavens, G., Cheon, Y.: Design by Contract with JML (2003)
39. Leroy, X.: The Objective Caml System Release 3.09. Institut National de Recherche en Informatique et en Automatique (2005)
40. Malayeri, D., Aldrich, J.: CZ: multiple inheritance without diamonds. In: Proc. OOPSLA 2009, pp. 21–40 (2009)
41. Meyer, B.: An Eiffel Tutorial. Technical Report TR-EI-66/TU, ISE (2001)
42. Mikhajlov, L., Sekerinski, E.: A Study of the Fragile Base Class Problem. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 355–382. Springer, Heidelberg (1998)
43. Moon, D.A.: Object-oriented programming with Flavors. In: Proc. OOPSLA 1986, pp. 1–8. ACM Press (1986)
44. Nierstrasz, O., Ducasse, S., Reichhart, S., Schärli, N.: Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, University of Bern, Switzerland (2005)
45. Odersky, M., Altherr, P., Creme, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The Scala Language Specification, version 1.0. Technical report, Programming Methods Laboratory, EPFL (2006)
46. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable Units of Behaviour. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
47. Smith, C., Drossopoulou, S.: *Chai*: Traits for Java-Like Languages. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 453–478. Springer, Heidelberg (2005)
48. Stroustrup, B.: The C++ programming language, 3rd edn. AT&T (1997)

Marco: Safe, Expressive Macros for Any Language*

Byeongcheol Lee[1], Robert Grimm[2],
Martin Hirzel[3], and Kathryn S. McKinley[4,5]

[1] Gwangju Institute of Science and Technology
[2] New York University
[3] IBM Watson Research Center
[4] Microsoft Research
[5] The University of Texas at Austin

Abstract. Macros improve expressiveness, concision, abstraction, and language interoperability without changing the programming language itself. They are indispensable for building increasingly prevalent multilingual applications. Unfortunately, existing macro systems are well-encapsulated but unsafe (e.g., the C preprocessor) or are safe but tightly-integrated with the language implementation (e.g., Scheme macros). This paper introduces Marco, the first macro system that seeks both encapsulation and safety. Marco is based on the observation that the macro system need not know all the syntactic and semantic rules of the target language but must only directly enforce some rules, such as variable name binding. Using this observation, Marco off-loads most rule checking to unmodified target-language compilers and interpreters and thus becomes language-scalable. We describe the Marco language, its language-independent safety analysis, and how it uses two example target-language analysis plug-ins, one for C++ and one for SQL. This approach opens the door to safe and expressive macros for any language.

1 Introduction

Macros enhance programming languages without changing them. Programmers use macros to add missing language features, to improve concision and abstraction, and to interoperate between different languages and with systems. With macros, programmers use concrete syntax instead of tediously futzing with abstract syntax trees or, worse, creating untyped and unchecked strings. For instance, Scheme relies heavily on macros to provide a fully featured language while keeping its core simple and elegant. Programmers use the C preprocessor to derive variations from the same code base (e.g., with conditional compilation) and abstract over the local execution environment (e.g., defining types and variables in system-wide header files). Web programmers use macros in PHP and similar languages to generate HTML code. Programmers also use macros

* This research was supported by the Samsung Foundation of Culture, and NSF grants CCF-1018271, CCF-1017849, and SHF-0910818.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 589–613, 2012.

to generate strings containing SQL queries that interoperate with databases. As illustrated by they last two examples, macros do not only improve individual languages, but are also indispensable for building increasingly prevalent multilingual applications.

Programmers typically write macros in a *macro language* and the macro system generates code in a *target language*. Programmers embed macros in a *host language*. The host and target languages may differ. Macros are resolved before the target-language code is compiled or interpreted. Macro systems fall into two main categories. (1) Some macro systems are well encapsulated from the compiler or interpreter but are unsafe, e.g., the C preprocessor executes before the target-language compiler, but may generate erroneous code. (2) Some macro systems are safe but are tightly integrated with the target languages, e.g., the Scheme interpreter hygienically implements the core language together with its macro system [13,5].

Neither option is particularly attractive. Well encapsulated but unsafe macro systems lead to buggy target code. Notably, the C preprocessor operates on tokens, is not guaranteed to produce correct code, and C programs with macros consequently contain numerous errors [6]. Safe but tightly-integrated macro systems are limited to a prescribed combination of host and target languages and cannot be shared across languages. Developers must learn macro programming for every host and target language combination. Likewise, language designers need to design and implement macros for every combination. For example, while JSE [2] adapts Dylan's macros [20] to Java, it requires design changes and a fresh implementation. This burden increases the temptation to omit macros or use an encapsulated but unsafe macro system, such as GNU M4 [15]. Given the utility of macros and the diversity of current and future languages, expressive safe macros that scale across host and target languages are clearly desirable.

This paper introduces the Marco macro system, which delivers encapsulation and safety, making macros *language scalable*. Our key insight is that the macro system does not need to implement every target-language rule for target-language safety, but rather, it can reuse off-the-shelf target-language compilers or interpreters to do the job. Specifically, prior work on *syntactically* safe macros required the macro system to have a target-language grammar; this paper shows how to enforce syntax safety without that. Similarly, prior work on macros with safe *naming discipline* required the macro system to implement target-language scoping rules; this paper shows how to enforce naming discipline without reimplementing those either. Consequently, Marco composes a language-independent macro translator with unmodified target-language compilers and interpreters that check for most rule violations. The only requirement on the target-language processors is that they produce descriptive error messages that identify locations and causes of errors.

We designed Marco to meet three criteria: expressiveness, safety, and language scalability. For expressiveness, the Marco language has static types, conditionals, loops, and functions, making it Turing-complete. Marco supports target-language *fragments* as first-class values. As indicated by the name, fragments

need not be complete target-language programs. Rather, they contain portions of a target-language program along with *blanks* that other fragments fill in. For safety, Marco uses macro-language types to check target-language syntax, and uses dataflow analysis to check target-language naming discipline. For language scalability, Marco relies on error messages from target-language processors, making it the first safe macro system that is independent of the target language. We demonstrate Marco for two target languages. For depth, we chose C++, which is relatively complex due to its rich syntax, types, and many other features. For breadth, we chose SQL, which differs substantially from C-like languages and is of critical importance to many web applications. Marco currently checks for syntactic well-formedness and naming discipline; we leave type checking target-language code for future work. Furthermore, Marco programs are currently stand-alone; we leave host-language integration for future work, e.g., by using compositional techniques from Jeannie for C and Java [10].

In summary, this paper's contributions are: (1) The design of a safe and expressive macro language that is scalable over target languages. (2) A macro safety verification mechanism that uses unmodified target-language compilers and interpreters. (3) An open-source implementation[1] of the target-language independent macro processor and plug-ins for C++ and SQL.

2 Marco Overview

Fig. 1 illustrates the Marco architecture. It shows the Marco *static checker* taking a Marco program as input and verifying that target-language fragments are correct at macro definition time. Because Marco supports code generation based on external inputs, including additional target-language fragments, some syntax and/or name errors may survive static checking undetected. Consequently, the Marco *dynamic interpreter* verifies fragments again at macro instantiation time, after Marco fills in all blanks. This double checking is critical for developing macro libraries, where one group of programmers writes the macros and another group instantiates them with application-specific inputs. Both the static checker and dynamic interpreter rely on common *oracles* to verify target-language code syntax and naming discipline and detect any errors. The oracles abstract over the different target languages. They query a target-language

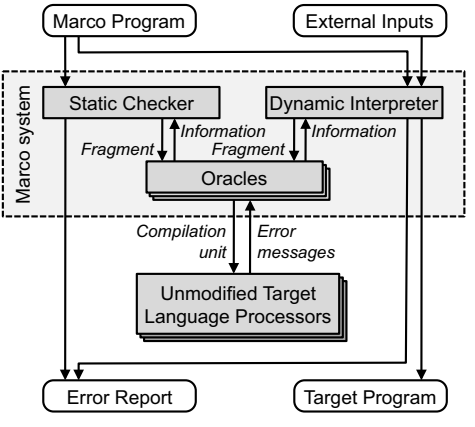


Fig. 1. The Marco architecture

[1] Available at http://cs.nyu.edu/xtc/

processor (currently, gcc for C++ macros and SQLite for SQL macros) by submitting specially crafted small compilation units.

Marco achieves language scalability by composing off-the-shelf target-language compilers and interpreters with a common translation engine. To add a new target language, developers implement a simple lexical analysis module that recognizes target-language identifiers and the end of target-language fragments. They also implement a plug-in with three oracles that (1) check for syntactic well-formedness, (2) determine a fragment's free names, and (3) test whether a fragment captures a given name. Everything else is target-language independent. In particular, Marco includes a reusable dataflow analysis, which propagates free and captured identifiers and reports accidental name capture.

3 The Marco Language

This section describes the Marco language, using examples, grammar rules, and type rules. The Marco language is a statically typed, procedural language. It supports macros using three constructs: code types, fragments, and blanks.

The example *synch* C++ macro in Fig. 2 ensures that lock acquire and release operations are properly paired (modulo exceptions), which C++ does not ensure. Lines 1–3 contain the signature of the Marco function *synch*, which takes two parameters, a C++ identifier and a C++ statement, and returns a C++ statement. The **code** type is parameterized by the target language and a nonterminal. Line 5 uses the back-tick operator (`) to begin a *fragment*, which is a quoted piece of target-language code. Line 6 uses the dollar operator (**$**) to begin a *blank*, which is an escaped piece of Marco code embedded in a fragment. The evaluation rule for a fragment first evaluates embedded blanks, then splices their results into the fragment's target-language code:

```
1 code<cpp,stmt>                    # code type
2 synch(code<cpp,id> mux,
3       code<cpp,stmt> body) {
4   return
5     `cpp(stmt)[{                  # C++ fragment
6       acquireLock($mux);
7       $body                       # blank
8       releaseLock($mux);
9     }];
10 }
```

Fig. 2. Marco code to generate C++

$$\frac{\forall i \in 1 \ldots n : \mathsf{Env} \vdash e_i \longrightarrow \beta_i}{\mathsf{Env} \vdash \text{`}lang(nonT)\,[\alpha_0 \$ e_1 \alpha_1 \ldots \$ e_n \alpha_n] \longrightarrow \text{`}lang(nonT)\,[\alpha_0 \beta_1 \alpha_1 \ldots \beta_n \alpha_n]} \text{(E-FRAGMENT)}$$

Each α_i is a sequence of target-language tokens, each $\$e_i$ is a blank, and each β_i is the result of evaluating a blank to a sequence of target-language tokens. The result is the concatenation of all α_i and β_i.

Fig. 3 presents the Marco grammar. The interesting grammar rules are *fragment* and its helpers. A fragment, such as `` `cpp(stmt)[...$x...$y...] ``, consists of a head and a sequence of fragment elements. The head specifies the target language, a nonterminal, and an optional list of captured identifiers. We use this list when checking the target code's naming discipline (see Section 7).

```
program      ::= functionDef+
functionDef  ::= type ID '(' (formal (, formal)*)? ')' '{' stmt* '}'
formal       ::= type ID
stmt         ::= '{' stmt* '}'                                  # block
               | type ID '=' expr ';'                           # variable declaration
               | 'if' '(' expr ')' stmt ('else' stmt)?          # conditional
               | 'for' '(' ID 'in' expr ')' stmt                # loop
               | 'return' expr ';'                              # function return
               | expr ';'                                       # expression statement
expr         ::= fragment                                       # fragment
               | '(' expr ')'                                   # parentheses
               | ID                                             # variable use
               | expr INFIX_OP expr                             # infix operation
               | ID '(' (expr (, expr)*)? ')'                   # function call
               | expr '.' ID                                    # record attribute
               | expr '[' expr ']'                              # list subscript
               | '[' (expr (, expr)*)? ']'                      # list literal
               | '{' ID '=' expr (, ID '=' expr)* '}'           # record literal
               | 'true' | 'false' | INT | STRING                # primitive literal
type         ::= 'code' '<' language ',' nonTerm '>'            # fragment type
               | 'list' '<' type '>'                            # list type
               | 'record' '<' formal (, formal)* '>'            # record type
               | 'boolean' | 'int' | 'string'                   # primitive type
fragment     ::= fragmentHead '[' fragmentElem* ']'
fragmentHead ::= '`' language '(' nonTerm (',' capture)? ')'
language     ::= ID
nonTerm      ::= ID
capture      ::= 'capture' '=' '[' ID (, ID)* ']'
fragmentElem ::= TARGET_TOKEN | blank
blank        ::= '$' baseExpr
```

Fig. 3. Marco grammar

There are two kinds of fragment elements in Marco: target-language tokens and blanks. Since Marco identifies fragment elements with square brackets, the Marco parser must count matching square brackets in the fragment itself to find the end, e.g., in `` `cpp(expr)[arr[idx]] ``. It should, however, ignore square brackets that appear in target-language strings or comments, e.g., in `` `cpp(expr)[printf("[")] ``. To enforce the naming discipline, the Marco parser must find the fragment's identifiers. It should not treat a target language's keywords or numerical suffixes as identifiers, e.g., in `3.1e4` or `111u`. Since different languages have different keywords, literals, and comments, Marco must be configured with target-language specific lexers. To select lexers based on the syntactic target-language context identified by the fragment's head, we use the *Rats!* [8] scannerless and modular parser generator. The corresponding target-language lexers are very simple and recognize only the target-language tokens listed above.

The static type system includes the primitive types **`boolean`**, **`int`**, and **`string`**; **`list`** parameterized by element type; **`record`** parameterized by attribute names and types; and **`code`** parameterized by target language and nonterminal.

The latter is key to *target language scalability*. For example, Fig. 4 shows a Marco macro generating SQL. Compared to Fig. 2, the Marco syntax and semantics remain the same, but the target language and therefore the code type parameters differ. The Marco engine uses code types to invoke the appropriate target-language oracles, which determine syntactic well-formedness as well as free and captured identifiers. It tracks code types to check syntax when filling in blanks in fragments. And it tracks names during its dataflow analysis, which ensures that identifiers from multiple macros generate consistent bindings. In other words, the strongly typed quote and unquote mechanism lets us maximize Marco's target-language independent functionality while delegating checking to the target-language specific oracles.

```
1 code<sql,query>
2 genTitleQueryInSQL(code<sql,expr> pred){
3   return `sql(query)[select title
4       from moz_bookmarks where $pred
5     ];
6 }
```

Fig. 4. Marco code to generate SQL

The type rule for a Marco fragment first checks the types for each of the embedded blanks, which must result in code belonging to the same target language (*lang*). It then uses the language *lang*, the nonterminal *nonT* of the fragment, the nonterminals $nonT_i$ of each of the blanks, and the contents of the fragment as inputs to a syntax oracle. As far as the Marco type system is concerned, the syntax oracle is a black-box that either succeeds or fails. If the oracle succeeds, the type of the fragment is **code**<*lang*, *nonT*>.

$$\frac{\forall i \in 1 \ldots n : \Gamma \vdash e_i : \textbf{code}\langle lang, nonT_i\rangle \quad syntaxOracle(lang)(nonT, [nonT_1, \ldots, nonT_n], \alpha_0 \$1 \alpha_1 \ldots \$n \alpha_n)}{\Gamma \vdash \text{`}lang(nonT)\,[\,\alpha_0 \$e_1 \alpha_1 \ldots \$e_n \alpha_n\,] : \textbf{code}\langle lang, nonT\rangle} \quad \text{(T-Fragment)}$$

4 The Marco Analysis Framework

At its core, the Marco system provides a static checker and a dynamic interpreter. The static checker verifies correctness at macro development time. The dynamic interpreter generates target-language code and verifies correctness at macro instantiation time. The two components share target-specific oracles, which check syntactic well-formedness and naming discipline in target-language fragments.

Each *oracle* mediates between the target-language independent Marco engine and an off-the-shelf target-language *processor*, i.e., compiler or interpreter. The oracle converts fragments into compilation units, passes the compilation units to the target-language processor, and then parses any error messages in the output. The only target-language specific parts newly developed for Marco are the target-specific lexers (see Section 3) and oracles, which reuse existing target-language processors. A key advantage of Marco over other safe macro systems is that it does not require new or even modified target-language processors.

In more detail, each target-language plug-in provides three oracles: *syntax*, *free-names*, and *captured-name*. Marco itself is implemented in Java, and each

target-language plug-in implements the same three Java interfaces. Since Java already integrates database access through JDBC, the SQL plug-in uses this API. In contrast, the C++ plug-in interacts with gcc through the file system. Either way, all target-language interactions share two characteristics. First, target-language processors receive programs as sequences of characters: strings for JDBC and files for gcc. In other words, we lower the tokenized fragments into character strings. Second, the processor outputs are strings that indicate syntactic or semantic errors, which are then parsed by the plug-in. At the same time, the concrete error reporting mechanism depends on the target language, e.g., Java exceptions for JDBC and standard error output for gcc.

To check target-language syntax, the system first parses a Marco program and tokenizes the target-language fragments. It then ensures that the target-language fragments are consistent with their declared **code** type parameters. For example, consider `` `cpp(expr)[x = 1;] ``. The Marco type checker applies rule T-Fragment from Section 3, which triggers a call to the C++ syntax oracle: *syntaxOracle*(`cpp`)(`expr`, [], '`x = 1;`'). The C++ syntax oracle then generates the following compilation unit for the unmodified C++ compiler, i.e., gcc: `int check_expr(){return (x = 1;);}` For this input, gcc reports an error complaining about the spurious semicolon after `x = 1`. Based on this error message, the oracle deduces that the fragment was syntactically ill-formed for nonterminal `expr`. Since the oracle fails, Marco type-checking fails, and the system reports an error. This example ignores idiosyncrasies of blanks and C++, which Sections 5 and 6 explore.

For naming disciplines, consider a fragment f_1 that fills a blank in fragment f_2. Fragment f_1 is `` `sql(expr)[birthYear >= 1991] ``, and fragment f_2 is `` `sql(query)[select name from Patrons where $pred] ``. Using the SQL free-names oracle, Marco discovers that f_1 contains the free identifier `birthYear`. It uses dataflow analysis to discover that f_1 flows into the blank in f_2. Finally, Marco uses the SQL captured-name oracle to check if identifier `birthYear` is captured by blank `$pred`. Programmers use annotations to tell Marco when a capture is intentional; otherwise, Marco reports an accidental-capture error. Subsequent sections describe how the oracles turn errors from target-language processors into information for Marco's static checker and dynamic interpreter.

5 Checking Syntactic Well-Formedness

This section describes how Marco checks target-language syntax. The *syntax oracle* is the interface between the target-language agnostic Marco system and the black-box target-language processors. The signature of the syntax oracle, as embodied in type rule T-Fragment from Section 3, is:

$$\mathit{syntaxOracle} : \mathit{lang} \to (\mathit{nonT}, \mathbf{list}\langle \mathit{nonT} \rangle, \alpha_0 \$1 \alpha_1 \ldots \$n \alpha_n) \to \mathbf{list}\langle \mathit{error} \rangle$$

For example, consider the following invocation of the syntax oracle:

$$\mathit{syntaxOracle}\ (\ \mathit{sql}\ )\ (\ \mathit{query}, [\mathit{expr}], \text{`}\mathbf{select}\ a\ \mathbf{from}\ B\ \mathbf{where}\ \$1\text{'}\ )$$

Table 1. Helper fragments for syntax oracles. Place-holder fragments fill in blanks. Completion fragments turn a fragment into a self-contained compilation unit. Marco fills in **$***fresh* blanks with fresh identifiers, and **$***orig* blanks with the original fragment.

Marco type	Place-holder fragment	Completion fragment
code<*sql,expr*>	`0`	**select** $*fresh1* **from** $*fresh2* **where** (**$***orig*)
code<*sql,query*>	**select** *	**$***orig*
code<*sql,qlist*>	*/*empty*/*	**$***orig*
code<*cpp,expr*>	`0`	**int** $*fresh*() { **return** (**$***orig*); }
code<*cpp,stmt*>	`;`	**void** $*fresh*() { **if**(1) **$***orig* **else**; }
code<*cpp,id*>	**$***fresh*	**int** $*fresh*() { **return** (**$***orig*); }
code<*cpp,type_sp*>	**int**	**$***orig* $*fresh*;
code<*cpp,type_id*>	**int**	**int** $*fresh*() { **return** *sizeof*(**$***orig*); }
code<*cpp,fdef*>	**void** $*fresh*(){}	**$***orig*
code<*cpp,mdecl*>	**int** $*fresh*;	**class** $*fresh* { $*orig* };
code<*cpp,decl*>	**int** $*fresh*;	**$***orig*
code<*cpp,cunit*>	*/*empty*/*	**$***orig*

In this example, the target language is SQL, the nonterminal of the fragment is *query*, and there is one blank, whose nonterminal is *expr*. The fragment contents have the form α_0**$**1$\alpha_1$, where α_0 is the token sequence before the blank, **$**1 marks the location of the blank, and α_1 is the token sequence after the blank. In the example, α_0 is '**select** *a* **from** *B* **where**' and α_1 is empty. The remainder of this section describes the syntax oracle algorithm for producing compilation units, interpreting the results, and iterating when necessary.

5.1 Syntax Oracle Algorithm

The syntax oracle algorithm has four steps. The key challenge is to fill in each *blank* in the target-language fragment.

Step 1: Fill in blanks. The syntax oracle starts by filling in each blank with a place-holder fragment. The middle column of Table 1 shows the place-holder fragments for each code type in Marco's SQL and C++ plug-ins. Each such place-holder fragment is syntactically valid for a given nonterminal. In the example above, the nonterminal for blank 1 is *expr*, so the syntax oracle fills in blank 1 with the place-holder fragment for SQL expressions, which is `0`. The result is the fragment **select** *a* **from** *B* **where** `0`. Intuitively, filling in blanks with fixed fragments works because target languages have (more or less) context-free grammars, and the syntax oracle can check syntactic validity even when there are semantic errors. In the example, the place-holder fragment is of type integer and the blank expects type boolean, but this semantic mismatch is irrelevant to syntactic well-formedness.

Step 2: Complete the fragment. The syntax oracle completes fragments to obtain self-contained compilation units. In the example, the fragment is already a full query. The right column of Table 1 shows the completion fragments for each of the code types in Marco's SQL and C++ plug-ins. In addition to turning a fragment into a compilation unit, Step 2 generates boiler-plate syntax. For SQL,

it adds code to begin and then abort a transaction, which prevents side-effects from sending the SQL query to a live database during analysis.

Step 3: Run the target-language processor. The syntax oracle next sends the completed fragment to the target-language processor and collects any error messages. For SQL, Marco makes a JDBC call and catches any exceptions. For C++, Marco generates a file with the fragment, compiles it with gcc, and reads any error messages from `stderr`.

Step 4: Determine oracle results. Finally, the syntax oracle translates errors from the target-language processor into oracle results. It must distinguish syntax errors from other errors, as a fragment only fails the syntactic well-formedness test if there are syntax errors. In C++, other errors may mask syntax errors, so the oracle may iterate to determine if the fragment also has a syntax error, as explained in Section 6. If the syntax oracle fails, the oracle maps error message line-numbers back to the original Marco code, and reports the error.

5.2 Syntax Oracle Example

Consider the example fragment `` `sql(expr)[type = ] ``, which is missing its right operand. Type rule T-FRAGMENT invokes the syntax oracle as follows: *syntaxOracle*(`sql`)(`expr`, [ ], '`type =`'). The oracle goes through its four steps:

1. Fill in blanks. This step is a no-op, since there are no blanks.
2. Complete the fragment. The oracle consults Table 1 to find the completion fragment for **code**<`sql,expr`>, yielding **select** *x* **from** *T* **where** (`type =`).
3. Run the target-language processor. The oracle uses JDBC to send the completed fragment to SQLite, and then catches the resulting SQLException, which contains the error message "Syntax error near '='.".
4. Determine result. Since the error from the target-language processor was a syntax error, the oracle reports this error back to the user.

Assume the user fixes the fragment, writing `` `sql(expr)[type = 1] ``, and then runs Marco again.

1. Fill in blanks. This step is still a no-op, since there are no blanks.
2. Complete the fragment yields **select** *x* **from** *T* **where** (`type = 1`).
3. Run the target-language processor. If there is no table *T* with an attribute `type` in the database, the error message is "No attribute '`type`' in table '*T*'.".
4. Determine result. Since the error is not a syntax error, the oracle succeeds and indicates that the fragment is syntactically well-formed.

5.3 Discussion

Good completion fragments (see Table 1) satisfy three properties: they are complete, conservative, and accurate. A *complete* fragment yields a complete compilation unit in the target language. A *conservative* fragment has a blank that accepts all fragments conforming to the original nonterminal. An *accurate* fragment rejects fragments that do not conform to the original nonterminal. Of these three properties, the first two help avoid spurious error messages, i.e., false positives, while the third helps avoid missed syntax errors, i.e., false negatives.

Consider the completion fragment for a C++ expression. One complete and conservative solution would be `int $fresh(){ return $orig; }`. This completion fragment is not accurate, since it accepts `0;`, which is a C++ statement and not a C++ expression. To increase accuracy, Marco adds parentheses around the blank, as in `int $fresh(){ return ($orig); }`. As another example, consider the completion fragment for a C++ statement. One complete and conservative solution is `void $fresh(){ $orig }`. However, this completion is not accurate, since it accepts `x=1;y=2;`, a sequence of two statements instead of a single C++ statement. Marco resolves this problem by inserting a conditional statement around the blank: `void $fresh(){ if(1) $orig else; }`. In our experience, enclosing fragments and blanks in delimiters or embedding them in other target-language constructs makes completion fragments more accurate.

6 Context-Sensitive Syntax

Most programming languages, including SQL, Java, ML, and Scheme, have context-free syntax. In this case, the syntax oracle from Section 5 works directly as described. It checks the syntactic well-formedness of a fragment in isolation based on the declared language and nonterminal. However, our goal is to handle any language, including languages with context-sensitive syntax such as C++. Prior work on safe macro systems does not address this issue. This section extends our syntax oracle to correctly deal with context sensitivity.

6.1 Context-Sensitive Syntax Examples

As a first example, consider the following C++ fragment:

```
`cpp(mdecl)[void* method(typeless o) { return 0; }]
```

The nonterminal *mdecl* stands for a member declaration. The fragment is syntactically well-formed for this nonterminal, since a method is a special case of a member. However, after using the completion fragment for *mdecl* from Table 1, gcc reports the following errors:

```
error: expected ';' at end of member declaration
error: expected ')' before 'o'
```

These are syntax errors, even though the root cause is a semantic problem: identifier *typeless* has not been declared as a type. When gcc cannot parse *typeless* as a declaration specifier, it speculates that *method* is a variable name. But the downstream tokens make no sense for a variable declaration. This case shows how a semantic problem, the missing declaration context for *typeless*, *induces* a syntax error.

To resolve such cases, our C++ syntax oracle enumerates all identifiers in the input fragment, and speculates one by one that they are type names (i.e., the opposite of gcc's speculation). In the example, the syntax oracle finds three identifiers: *method*, *typeless*, and *o*. At first, the syntax oracle speculates that

method is a type, but declaring it as such does not advance the location of the first error message. Then, the oracle speculates that *typeless* is a type and issues the following query to gcc:

```
class typeless { };                          // speculative context
class id1 {                                  // completion fragment
  void* method(typeless o) { return 0; } // input fragment
};
```

Since gcc reports no syntax errors for this query, the syntax oracle correctly concludes that the input fragment is syntactically well-formed and succeeds. In theory, a fragment with n ambiguous identifiers may require up to 2^n speculative queries for determining syntactic well-formedness. However, in practice, our simple heuristic of always advancing the location of the first reported error avoids this exponential explosion.

As a second example, consider another C++ fragment (based on [19]):

```
`cpp(stmt)[ A(*x)[4] = y; ]
```

The C++ compiler can parse this in two ways: either as a variable declaration or as an expression statement. If `A` is a type, then the code declares variable `x` as a pointer to an array of 4 elements of type `A`, and initializes it to `y`. On the other hand, if `A` is a function name, then the code calls the function with parameter `*x`, accesses element `[4]` of the result, and assigns it `y`. The ambiguity between declarations and statements is so prevalent in C++ that the language specification has a disambiguation rule of preferring declarations over expression statements [23, p. 802]. Though, C++, unlike C, does treat declarations as statements for its syntax.

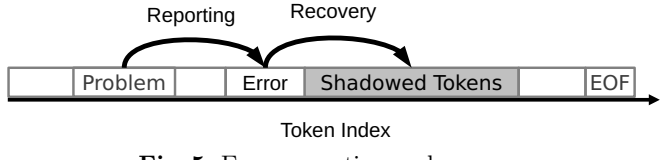


Fig. 5. Error reporting and recovery

If part of a program is not well-formed, language processors report the error and try to recover, so that they can report more than one error per invocation. Fig. 5 depicts how language processors scan through lexical tokens, detect a syntactic or semantic problem, generate an error message, skip several tokens, and continue analysis. For instance, ANTLR-generated parsers report syntax errors and then recover by either inserting a token or skipping several downstream tokens. The hand-written parsers in gcc report both syntax and semantic errors, and may skip all or some downstream tokens.

If the skipped tokens contain a syntax error, then the error recovery for the first error *shadows* the syntax error. Therefore, the absence of syntax errors does not imply syntactic well-formedness. The example fragment `A(*x)[4] = y;` triggers the following error messages:

```
error: 'x' was not declared in this scope
error: 'A' was not declared in this scope
error: 'y' was not declared in this scope
```

These semantic errors may shadow downstream syntax errors, so our oracle speculates that an identifier in a semantic error may be either a type or variable name. In the example, making `A` a type name and `x` and `y` variable names eliminates the missing declaration errors, as shown in the following oracle query:

```
class A {}; class {} x; class {} y;  // speculative context
void query() {                       // completion fragment
  A(*x)[4] = y;                      // input fragment
}
```

This fragment still yields a semantic error ("cannot convert '<anonymous class>' to '`A (*)[4]`'"), but the error cannot shadow syntax errors. Hence, the oracle concludes that the fragment is syntactically well-formed.

6.2 Error Classification

The syntax oracle is concerned with syntax errors, and, in an ideal world, it would not have to deal with semantic errors. However, as demonstrated by the above examples, semantic problems affect syntax errors in two cases: a semantic problem can *induce* a syntax error, or a semantic problem can *shadow* a syntax error. In the induced-error case, it appears as if the fragment is syntactically ill-formed, but it is actually well-formed. In the shadowed-error case, it appears as if the fragment is syntactically well-formed, but it is actually ill-formed.

We need not handle syntax errors that induce or shadow another error; the first error suffices to conclude that the fragment is syntactically ill-formed. Neither do we need to handle the case of a semantic problem inducing or shadowing a semantic error; as long as that error in turn does not induce or shadow a syntax error, it does not affect the syntax oracle. Consequently, the syntax oracle must recognize two classes of errors: (1) syntax errors and (2) semantic errors that may shadow syntax errors. We systematically investigated all C++ errors generated by gcc to validate that our syntax oracle handles these cases correctly. Our investigation found that there are two kinds of syntax errors: parsing errors, which are generated while parsing, and post-parsing errors, which are generated after parsing but still capture violations of syntactic constraints such as **`case`** labels always appearing inside **`switch`** statements. Parsing errors are easy to recognize (they all begin with the word "expected" and include a token or nonterminal symbol), and there are only a few post-parsing errors.

To collect all shadowing error messages, we identified the seven error recovery routines in gcc that update the parser state to skip tokens until a synchronization token. For example, one such routine is `skip_until_sync_token()`. Next, we enumerated all call-sites for the recovery routines. We found that code leading up to these call-sites commonly looks as follows:

```
bool ok = perform_semantic_check();
if (ok)
  error("A");
else
  error("B");
if (!ok) {
  error("C");
  skip_until_sync_token();
}
```

If *ok* is false, the compiler invokes *skip_until_sync_token*() and thus skips tokens, which may contain syntax errors. Consequently, errors "B" and "C" may shadow syntax errors but error "A" may not. In most cases, we only had to look at a single routine to understand error shadowing, though in a few cases multiple routines were involved. We found that shadowing errors are most commonly lookup errors, which indicate that an identifier has not been declared; though a few non-lookup errors shadow other errors.

6.3 Iterative Syntax Oracles in Marco

In summary, context sensitivity prevents a Marco oracle from concluding syntactic well-formedness based solely on the absence of syntax errors. In particular for C++, a semantic problem can either induce or shadow a syntax error. Therefore, Marco's syntax oracle for C++ speculatively resolves syntax errors and shadowing semantic errors by issuing repeated queries to gcc, each with a different speculative context. In other words, the oracle speculates declarations for identifiers as variables or types. If Step 4 from Section 5.1 detects semantic errors, the algorithm iterates back to Step 2 and resolves them by enumerating the possible declarations for identifiers.

7 Checking Naming Discipline

This section describes how Marco uses dataflow analysis to ensure that macro expansion does not cause accidental name capture in the target language. Some macro systems, notably Scheme, prevent capture by automatically renaming variables, but that requires deep target-language specific knowledge and is therefore not an option for Marco, which is target-language agnostic. Accidental name capture is a typical bug when using the C preprocessor, as illustrated in Fig. 6.

```
1 #define swap(v,w) {int temp=v; v=w; w=temp;}
2 int temp = thermometer();
3 if (temp<lo_temp) swap(temp, lo_temp)
```

Fig. 6. Example for accidental name capture bug with C preprocessor [5,6]

Line 1 declares a macro *swap*, which uses a local variable *temp* standing for "temporary." Line 2 declares a different variable *temp* standing for "temperature" that is outside the scope of the macro. Line 3 passes the name *temp* as an

actual parameter to the formal `v` of `swap`. This use of `swap` captures the name `temp`. Since the code outside the macro uses `temp` for "temperature" and not "temporary," the name capture is called *accidental*.

More generally, accidental name capture happens when a first fragment f_1 contains a free name x; a second fragment f_2 unintentionally captures name x at blank b; and f_1 flows into b. Marco detects capture as follows. The *freeNamesOracle* discovers all free names in fragment f_1. Marco's forward dataflow analysis propagates free names to blanks. The *capturedNameOracle* discovers whether f_2 captures name x at blank b. In dataflow terminology, the analysis state is a map from meta-language variables to free target-language names. The GEN-set of a fragment is the set of free names in the fragment. The KILL-set of a fragment at a blank is the set of names that it captures at that blank.

Marco uses static dataflow analysis at macro definition time and dynamic dataflow analysis in the interpreter at macro instantiation time. The oracles are target-language specific and use the off-the-shelf target-language processor as a black-box to generate error messages that reveal information about free and captured names. The dataflow analysis itself is target-language independent.

7.1 Free-Names Oracle

The signature of the free-names oracle is:

$$\mathit{freeNamesOracle} : \mathit{lang} \rightarrow (\mathit{nonT}, \mathbf{list}\langle \mathit{nonT} \rangle, \alpha_0 \$1 \alpha_1 \ldots \$n \alpha_n) \rightarrow \mathbf{list}\langle \mathit{ID} \rangle$$

For example, given fragment `` `cpp(expr)[100 * (1.0 / (foo))] ``, Marco invokes the free-names oracle as follows:

$$\mathit{freeNamesOracle}(\mathit{cpp})(\mathit{expr}, [\,], \text{`}\texttt{100 * (1.0 / (foo))}\text{'})$$

A target-language name is free if it is not bound inside the fragment. In the example, `foo` is free, and the oracle should return the list [`foo`]. To obtain this result, the free-names oracle first consults Table 1 to instantiate a completion fragment that will be sent to gcc:

```
int query_expr() { return (100 * (1.0 / (foo))); }
```

For this query, gcc returns the error message "name '`foo`' was not declared in this scope." The free-names oracle looks exactly for this kind of error message, which indicates that a name is undefined. In the example, the oracle speculates that `foo` is free. To validate this hypothesis, it executes one more query. It prepends a declaration of the name `foo` to the compilation unit and sends it to gcc again. In the example, the test becomes:

```
int foo;
int query_expr() { return (100 * (1.0 / (foo))); }
```

This modification resolves the declaration error and confirms the hypothesis as correct. Hence, the oracle adds the name `foo` to the list of free names. It repeats this process until it does not observe any more declaration errors. In summary, Marco exploits the fact that a name in a fragment is free, as long as it can be bound by a declaration in the enclosing scope.

7.2 Captured-Name Oracle

The captured-name oracle checks if a target-language name is captured by a blank in a fragment and thus if it is safe to fill in the blank with a fragment in which the name is free. The signature of the captured-name oracle is:

$$\begin{aligned} \mathit{capturedNameOracle} : \ & \mathit{lang} \\ & \rightarrow (\, \mathit{nonT}, \mathbf{list}\langle \mathit{nonT} \rangle, \\ & \quad \alpha_0 \mathbf{\$}1 \alpha_1 \ldots \mathbf{\$}n \alpha_n, \mathbf{int}, \mathit{ID}) \\ & \rightarrow \mathbf{boolean} \end{aligned}$$

We check the blank number **int** and the free name *ID* for capture. Consider swapping two integers: `` `cpp(stmt)[{int temp=$v; $v=$w; $w=temp;}] ``. The following oracle call checks whether blank 1 captures name `temp`:

```
capturedNameOracle ( cpp )
                   ( stmt, [expr, expr, expr, expr],
                     `{int temp=$1; $2=$3; $4=temp;}', 1, temp )
```

Since blank 1 in the fragment does in fact capture the target-language name `temp`, the oracle returns **true** as expected. SQL's scoping rules differ from C++ and implement semantics similar to a **with**-statement. For example, consider the fragment `` `sql(query)[select name from Patrons where $pred] ``. The blank in this fragment captures any names referring to column names in the `Patrons` table in the database. Our *capturedNameOracle* algorithm handles both target languages and their scoping rules with the same approach.

Consider invoking the *capturedNameOracle* to check if free name x is captured at blank number i. Similar to the other oracles, the captured-name oracle fills in all blanks j with $i \neq j$ using the nonterminal-specific place-holders from Table 1. However, for blank i, our analysis hypothesizes that x is captured at the blank. To find counter-evidence, it places x in the blank, wrapping it in boiler-plate code as necessary for syntactic well-formedness. If the target-language processor reports an "x is unknown" error message, then the oracle concludes that x is not captured at blank i, and returns **false**. Otherwise, it returns **true**.

7.3 Intentional Capture

The dataflow analysis propagates free target-language names through variables referencing target-language fragments to blanks. It reports an error if a blank accidentally captures a free name. To determine whether a capture is accidental or intentional, the analysis uses the optional **capture** annotation in the *fragmentHead* clause of the Marco grammar (see Fig. 3). If a name is listed in the **capture** annotation, the capture is intentional, otherwise the capture is accidental and Marco reports an error.

For an example of intentional capture, consider the Marco function `boundIf` in Fig. 7. The function implements an if-statement that binds the value of the condition to `it`, with the express purpose of exposing it to the body, i.e., blank 2.

```
1 code<cpp,stmt> boundIf(code<cpp,expr> cond, code<cpp,stmt> body) {
2   return `cpp(stmt, capture=[it])[{
3       int it = $cond;          #blank 1
4       if (it) { $body }        #blank 2
5     }];
6 }
```

Fig. 7. Example for intentional name capture when using Marco to generate C++ code

The annotation **capture**=[*it*] in line 2 indicates that any such capture is intentional. This convention makes it possible to fill blank 2 with a fragment, such as *printf*("%d", *it*);, that contains a free identifier *it* and have the body's *it* refer to the macro's *it*, as declared on line 3. Marco's captured-names oracle still detects the capture of the identifier *it*, but the **capture** annotation suppresses the corresponding error message. As a special case, if a blank captures an identifier provided by a previous blank in a binding position, e.g., **int** $*index*;, Marco assumes that the capture is intentional. No annotation is necessary.

7.4 Dataflow Analysis

The interesting statements for the dataflow analysis are Marco statements with fragments and blanks. Fig. 8 shows the transfer-function for such statements. Given a Marco statement and an input analysis state *inState*, the transfer function computes an output analysis state *outState*. The analysis state only changes for the Marco location w assigned by the statement. The captured-name oracle from Section 7.2 checks whether free names from $inState(v_1)$ through $inState(v_n)$ are captured by the fragment. If they are captured and the capture is not intentional (the set difference *Captured* − *Intentional* is non-empty), the analysis

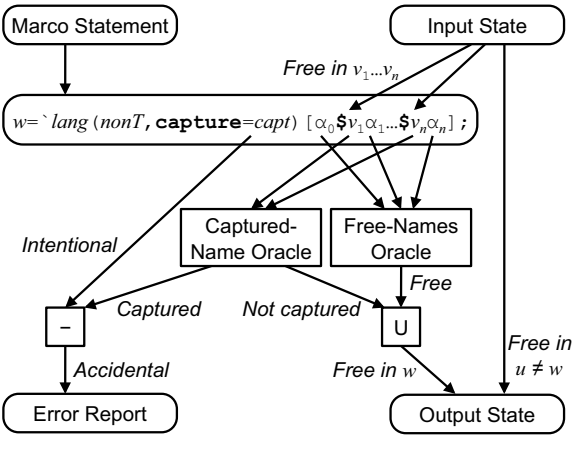


Fig. 8. Transfer functions for the naming-discipline analysis

reports an error. If they are not captured, then they are still free in w. In addition, the free-names oracle checks for free names in the constant portions α_0 thru α_n of the fragment. Those free names are also free in w. The resulting output state $outState(w)$ uses the free names for w as discovered by the oracles. For all other locations $u \neq w$, the transfer function forwards the free names from the input state $outState(u) = inState(u)$.

One pragmatic issue is how to report high-quality error messages in the case of accidental name captures. The analysis remembers which errors it has reported so far and avoids duplicates. Furthermore, the analysis tracks the originating fragment for each free name to more accurately report the source of accidental name captures. When the analysis detects an accidental capture, it reports both the line number of the origin and the line number of the capture.

Marco's static checker uses static dataflow analysis to enforce the naming discipline at macro definition time. It reports accidental name capture errors to macro authors. However, a Marco program may receive fragments as external inputs, and these fragments may contain free names. Consequently, Marco's interpreter uses dynamic dataflow analysis to enforce naming discipline again at macro instantiation time, now reporting accidental name capture errors to macro users. The dynamic dataflow analysis uses the same transfer function as the static analysis and it performs the same oracle queries.

8 Experimental Evaluation

This section experimentally validates the key characteristics of Marco: expressiveness, safety, and language scalability. To evaluate expressiveness, we implemented microbenchmarks from prior work and a code-generation template for a high-performance stream processing module. To evaluate safety, we execute Marco on each microbenchmark and on the stream processing code generator. To evaluate language scalability, we report statistics on the implementation effort for supporting different target languages.

8.1 Methodology

Tools and Environments. We use Marco r278 running on the Sun HotSpot Client JVM 1.6.0_21-ea. For the unmodified target-language processors, we downloaded and built gcc 4.6.1 as well as SQLiteJDBC v056 based on SQLite 3.4.14.2. We conducted all experiments on a Core 2 Duo 1.40 GHz with 4 GB main memory. The machine runs Ubuntu 11.10 on the Linux 3.0.0-12 kernel.

Marco Programs. We wrote 8 Marco microbenchmark programs with 22 macro functions derived from related work [5,26] and the `Aggregate` code generator derived from IBM InfoSphere Streams [16]. The `Aggregate` code generator produces C++ declarations, statements, and expressions that exercise classes, namespaces, and templates. Table 2 presents the microbenchmarks. The first four programs implement C++ macros from the MS^2 paper by Weise and Crew [26]. These macros add new abstractions such as resource management (`paint`), dynamic binding (`dynamic_bind`), rich exception handling (`exceptions`), and

Table 2. Oracle analysis results for the micro-benchmarks fragments

Marco Program	Fragment	Code Type	Size	Backtr.	Queries	Decls
paint	*Painting1*	**code**<*cpp*,*stmt*>	17	5	17	7
dynamic_bind	*dynamic_bind1*	**code**<*cpp*,*stmt*>	13	3	14	8
exceptions	*throw1*	**code**<*cpp*,*stmt*>	23	2	12	7
	throw2	**code**<*cpp*,*stmt*>	28	2	16	7
	catch1	**code**<*cpp*,*expr*>	1	1	3	3
	catch2	**code**<*cpp*,*stmt*>	51	1	8	4
	unwind_protect1	**code**<*cpp*,*expr*>	1	1	3	3
	unwind_protect2	**code**<*cpp*,*stmt*>	44	2	12	6
myenum	*myenum1*	**code**<*cpp*,*decl*>	5	0	1	1
	myenum2	**code**<*cpp*,*stmt*>	9	1	5	4
	myenum3	**code**<*cpp*,*decl*>	15	0	1	1
	myenum4	**code**<*cpp*,*stmt*>	14	2	8	6
	myenum5	**code**<*cpp*,*decl*>	18	0	2	2
discriminant	*discriminant1*	**code**<*cpp*,*expr*>	9	0	1	1
complain	*complain1*	**code**<*cpp*,*stmt*>	4	0	2	2
	main1	**code**<*cpp*,*expr*>	1	1	3	3
	main2	**code**<*cpp*,*stmt*>	13	0	2	2
swap	*swap1*	**code**<*cpp*,*id*>	1	1	3	3
	swap2	**code**<*cpp*,*id*>	1	1	3	3
	swap3	**code**<*cpp*,*stmt*>	28	5	31	12
SQLSyntax	*good1*	**code**<*sql*,*expr*>	3	0	1	0
	good2	**code**<*sql*,*stmt*>	6	0	1	0

multiple declarations (*myenum*). The next three programs implement C++ versions of the examples from "Macros That Work" by Clinger and Rees [5]. These macros illustrate naming issues during macro expansions. The final program generates SQL queries for extracting bookmark titles from a web browser's database.

Data collection methodology. To collect statistical results for fragment analysis, we turned on Marco's -pstat command-line option. To count source lines, we ran the sloccount utility. For the number of error handling rules, we manually examined source files in the Marco system.

8.2 Expressiveness and Safety

In Table 2, Column "Fragment" names the macros using logical function names. Column "Code Type" shows the types of the macros, which indicate the target language and nonterminal. Column "Size" counts the number of target-language tokens and blanks. The remaining columns present the results from running the oracle analysis. The oracle analysis synthesizes query programs to determine whether a fragment is syntactically correct. Column "Backtr." counts how often the syntax oracle needed to backtrack before finishing. Column "Queries" counts the number of compilation units sent to the target-language processor. Column

Table 3. Oracle analysis averages for the fragments in the *Aggregate* operator

Code type	Count	Size	Backtracks	Queries	Decls
code<*cpp*, *id*>	5	1.00	0.80	3.00	3.00
code<*cpp*, *type_spec*>	8	6.88	0.00	3.75	2.75
code<*cpp*, *type_id*>	1	1.00	0.00	2.00	2.00
code<*cpp*, *expr*>	12	4.50	0.08	2.67	2.58
code<*cpp*, *stmt*>	40	13.20	1.58	10.13	6.63
code<*cpp*, *fdef*>	11	31.00	4.09	21.73	9.36
code<*cpp*, *mdecl*>	22	12.36	0.05	3.23	3.00
code<*cpp*, *decl*>	13	11.38	0.00	4.08	3.00
code<*cpp*, *cunit*>	3	7.00	0.00	2.00	2.00

"Decls" shows the number of declarations synthesized by the oracle to provide evidence for syntactic well-formedness.

For the microbenchmark fragments, which contain 1–51 tokens or blanks, our oracle analyzer concludes syntactic well-formedness after evaluating 1–31 queries. The number of queries is proportional to the number of synthesized declarations rather than the size of input fragments. This result is not surprising, because the number of C++ parsing errors for syntactically well-formed fragments should be proportional to the number of undefined identifiers. About 20% of queries result in the oracle backtracking speculations.

This research was originally motivated by language interoperability and concision for IBM's InfoSphere Streams, a stream processing system [16]. A streaming application consists of data streams and operators. Each operator continuously consumes data from one or more input streams, performs its computation, and outputs one or more streams. An *Aggregate* operator uses sum, average, maximum, etc. over a sliding window and must be customized for the particular aggregate and data types. To implement these operator variants, InfoSphere Streams uses "code generation templates," i.e., macros that generate custom code for a specific operator variant. We re-implemented the code generation template for the *Aggregate* operator from InfoSphere Streams in Marco.

Table 3 presents average statistics for the 115 fragments in *Aggregate*. The first column classifies fragments by their code type and the second lists the number of fragments for each type. The remaining columns average the number of tokens and blanks ("Size"), the number of backtracks during oracle query analysis ("Backtracks"), the number of generated C++ compilation units for queries ("Queries"), and the number of helper declarations to disambiguate the C++ syntax ("Decls").

The *Aggregate* operator exercises more C++ specific code types than the micro-benchmarks. For instance, the 22 fragments of type **code**<*cpp*, *mdecl*>, where *mdecl* is the member-declaration nonterminal, generate C++ fields, methods, and constructors. No other macro system generates members of a C++ class and checks syntactic correctness of the generated code. Due to the ambiguity of C++ syntax, our oracle analyzer backtracked speculations 72 times over the

114 fragments. Most backtracking arises form C++ fragments that contain unknown identifiers or expressions statements. Even when an identifier is declared in C++, the gcc parser uses backtracking, so it comes as no surprise that our oracle also backtracks.

8.3 Language Scalability

To add target languages in a traditional safe macro system, the developer must modify the target-language processor, which is usually large and complex. To make matters worse, the modified target-language processor is effectively a branch version, and keeping it up-to-date with the main branch requires additional engineering effort. Adding a target language to Marco requires that the developer write a small plug-in consisting of a simple lexer and three oracles. The oracles wrap unmodified target-language processors. If these processors add or change their error messages, Marco must adapt. We argue that the effort is considerably lesser in the Marco approach.

C++ Plug-in. Like all target-language specific Marco plug-ins, our C++ plug-in consists of a lexical analyzer and three oracles. For the lexical analyzer, we define the *TARGET_TOKEN* terminal in Fig. 3 with a few lines of regular expressions for *identifier* (1), *literal* (5), *keyword* (74), and *preprocessing-op-or-punc* (72) [23]. Most regular expressions are trivial and only *identifier* and *literal* (6) require regular expression operators. Our three oracles consist of 1K+ non-blank source lines of Java. About half of the source lines implement oracle declaration queries, and the other half handle error messages. The error handlers contain 52 regular expressions to classify gcc error messages. In contrast, the gcc source files for the C++ front-end (`cc1plus`) contain 100K+ non-blank C source lines. The hand-written parser in `parser.c` has 14K+ non-blank source lines, and it relies on the semantic analysis in 96K+ non-blank source lines to disambiguate parsing decisions. Our C++ plug-in is much smaller and reuses, unmodified, the sophisticated code base that has been maintained for decades.

Fig. 9 presents an abstracted, static call graph for gcc's 1,400+ error messages. The cc1plus module consists of 53 C source files, 5,400+ procedures, and 67,000+ call sites. Out of 5,400+ procedures, 2,500+ procedures are reachable from the parser (c_parse_file). We use these 2,500+ procedures to over-approximate the syntactic and semantic error messages. We exclude the preprocessing library (libcpp) and backend library (libbackend) by treating them as terminal functions in the call graph. Our analysis assumes that all error messages must go through the three final error reporting functions: error, error_n, and error_at. We exclude the permerror function because it reports "permissive" errors that never shadow any downstream error messages. We label each node with a particular function name in a box or a group of functions in an oval. PARSING represents the functions for recognizing C++ nonterminals in the top-down parser. OTHERS represents the remaining functions. We label several edges with capital letters from A to P because their call sites tentatively characterize the kinds of error messages.

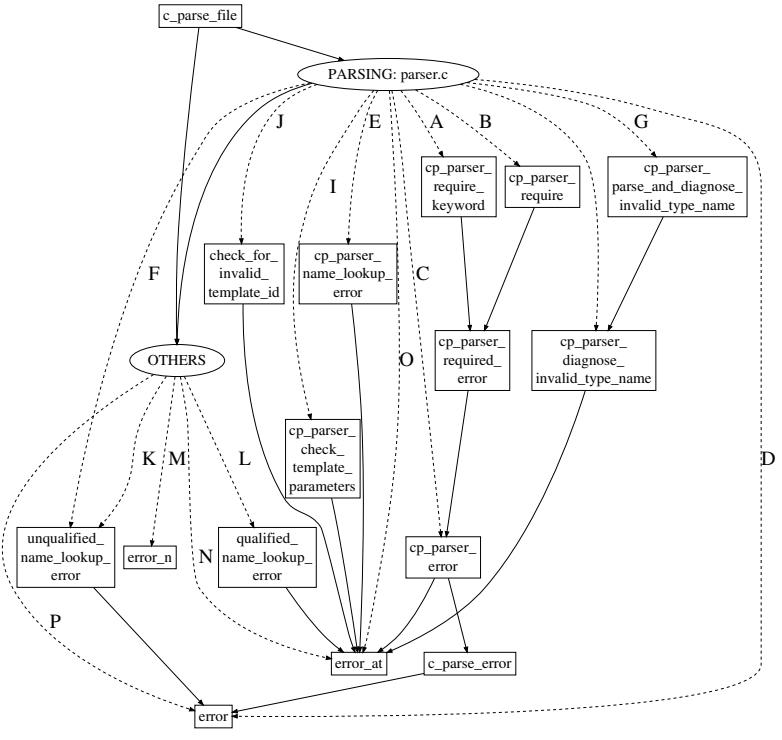


Fig. 9. Abstract Call Graph for error reporting routines

Table 4 maps the labeled call edges to our classifications of error messages. A and B are parsing syntax errors because they expect specific tokens including *keywords*, *punctuation*, and *operators* in C++. C contains 90 syntax errors and 2 semantic errors. E-L are semantic lookups identifying undeclared or unsatisfiable identifiers. D and M-P are mostly semantic errors. Overall, we identified 104 semantic errors that may shadow downstream error messages.

Our oracles recognize 384 critical error messages: 280 parsing error messages, 28 lookup error messages, and 76 other shadowing semantic error messages. A large fraction of these error messages are recognized by a few dozen regular expressions. For instance, all parsing error messages begin with `expected` and end with either a terminal or nonterminal symbol. The lookup error messages begin with `undeclared`.

SQL Plug-in. Our SQL plug-in consists of 40 lines for the lexical analyzer and 400 lines for the three SQL oracles. The lexical analyzer for SQL is simpler than the one for C++. Likewise, SQLite's parser is simpler than gcc's parser: it consists of about 1K source lines in `parser.y` written as an LALR specification. By using SQLite as a black-box language processor, Marco's SQL plug-in reuses not just the parser but also other components for checking naming discipline, all of which have been maintained and tested widely for over a decade.

Table 4. Mapping from calling contexts to error classes

Error Context	Call Sites	Syntax		Semantics		
		Parsing	Post-Parsing	Lookup	Other Shadow	Non-Shadow
A	27	27				
B	176	176				
C	92	73	17		1	1
D	22	3	2		17	
E	5			5		
F	2			2		
G	4			4		
H	2			2		
I	3			3		
J	4			4		
K	3			3		
L	5			5		
M	2					2
N	71					71
O	125	1			7	117
P	1,012				51	961

9 Related Work

Unlike previous macro systems, Marco is safe, well-encapsulated, and target-language agnostic at the same time. In the literature, safe macro systems are deeply coupled with the particular target language and its implementation (Section 9.1), whereas language-agnostic macro systems fail to provide safety guarantees (Section 9.2). Furthermore, while there is previous work that relies on error messages from unmodified language processors, this approach has not previously been applied to macro systems (Section 9.3).

9.1 Language-Specific Safe Macro Systems

Some macro systems check the safety of generated code by deeply coupling the macro language with the target language. Like Marco, these systems enforce macro safety, but, unlike Marco, they are target-language specific.

Syntax. To enforce syntactic well-formedness, previous safe macro systems usually rely on a grammar for the target language. For instance, MS^2 implements a C grammar [26], `metafront` implements grammars for Java and HTML [3], and Ur implements grammars for HTML and SQL [4]. These macro systems approach the level of complexity of extensible compiler toolkits such as ASF+SDF [25], Polyglot [18], or xtc [8,10]. While the approach of implementing a grammar works well enough for targeted language extensions and small domain-specific languages, it is problematic for large, existing languages. Besides the sizable development effort, another issue is compatibility. For example, HTML syntax is deceptively simple, but in practice, HTML processors have so many corner-cases that checkers resort to random testing [1]. Marco side-steps this issue by leveraging unmodified target-language processors for any language.

Scope. In the functional-languages community, a wide-spread technique for ensuring that macros respect scoping rules is *hygiene*. Kohlbecker et al. introduce

hygienic expansion [13]. Clinger and Rees present an improved algorithm for renaming identifiers to guarantee hygiene [5]. Kim et al. formally characterize accidental and intentional capture, but do not implement intentional capture [12]. All these systems depend on the syntax and scoping rules for the chosen target language. In contrast, Marco programmers declare intentional name capture and rely on the system to detect scoping violations with black-box target-language processors. Marco does not automatically rename identifiers, but rather uses dataflow analysis to detect and report errors on accidental name capture.

Semantics. Some macro systems check whether expanded fragments will pass type checking in the target language. Multi-stage extensions generate safe code in, for instance, ML [17], Haskell [21], and Java [27]. C++ concepts add contracts to templates [7]. MorphJ statically verifies some contracts so that expanded code will not have name-resolution conflicts [11]. Quail checks types between SQL queries and the database system [24]. However, target-language agnostic type checking is an open problem that has not been addressed by any of these systems, and we do not yet address it in Marco either.

9.2 Language-Agnostic Unsafe Macro Systems

The idea of macro systems that work for any target language dates back at least to GPM [22]. GPM is credited as an ancestor of M4, a general-purpose preprocessor widely installed on GNU platforms today [15]. Both offer target-language independence, but neither is safe. Perhaps the most used macro system today is the C preprocessor. Ernst et al. present an empirical study that demonstrates numerous violations of safety rules when using the C preprocessor [6]. Furthermore, code-generation based on concrete syntax is widely used in web applications. The PHP language is primarily a code generator for HTML and JavaScript. Programmers often generate SQL code by manipulating strings in general-purpose languages such as Java.

Compared to these systems, Marco adds safety checks while remaining expressive and language-agnostic. Marco relies on high-quality error messages from target-language processors. We believe that our reliance on descriptive error messages aligns well with compiler and interpreter writers who want to provide precise explanations for compilation and execution failures. Fragment code types constrain both the target-language and the nonterminal, enabling syntactic well-formedness checks in isolation. Marco is the first language-agnostic macro system to rely on a dataflow analysis for enforcing naming discipline.

9.3 Using Messages from Black-Box Compilers

A few previous systems rely on error messages from unmodified language processors. Notably, Seminal analyzes error messages from the OCaml and gcc compilers and suggests changes for ill-formed programs [14]. Autoconf compiles specially crafted C/C++ programs, analyzes any error messages, and determines if preprocessor symbols or header files are available in the build environment. The HelpMeOut system mines IDE logs to discover common bug fixes, and then

proposes them to programmers based on currently displayed error messages [9]. Similar to Marco, these systems execute unmodified language processors, and inspect the error messages for clues. Unlike Marco, none of these systems is a macro system. To our knowledge, Marco is the first system that mines error messages from black-box compilers and interpreters for safe code generation.

10 Conclusion

Macros that are expressive, safe, and language scalable at the same time have the potential to significantly improve programmer productivity, particularly for increasingly prevalent multilingual applications. This paper has presented the first such macro system called Marco. Our work is based on two key ideas. First, a plug-in facility provides target-language specific oracles implemented with off-the-shelf compilers and interpreters. In particular, we have identified three simple oracles: syntax, free-names, and captured-name. They are sufficient for ensuring syntactic correctness and naming discipline of macros. Oracles are discharged by submitting specially crafted programs to the target-language processor and then analyzing any resulting error messages. Second, a statically typed quote/unquote facility maximally exploits the target-language independent translation engine. Notably, the type of a fragment specifies its target language and its nonterminal, which the engine uses to invoke the appropriate language-specific oracles. Our evaluation of the Marco prototype supporting C++ and SQL demonstrates the viability of this approach. Future work should explore additional target languages and safety guarantees. Overall, our work demonstrates that safe code generation through macros is orthogonal to language implementation and can be well-encapsulated and language-scalable at the same time.

References

1. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in dynamic web applications. In: ACM International Symposium on Software Testing and Analysis, ISSTA (2008)
2. Bachrach, J., Playford, K.: The Java syntactic extender (JSE). In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2001)
3. Brabrand, C., Schwartzbach, M.I., Vanggaard, M.: The metafront system: Extensible parsing and transformation. Electronic Notes in Theoretical Computer Science 82(3) (December 2003)
4. Chlipala, A.: Ur: Statically-typed metaprogramming with type-level record computation. In: ACM Conference on Programming Language Design and Implementation, PLDI (2010)
5. Clinger, W., Rees, J.: Macros that work. In: ACM Symposium on Principles of Programming Languages, POPL (1991)
6. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering (TSE) 28(12) (December 2002)
7. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2006)

8. Grimm, R.: Better extensibility through modular syntax. In: ACM Conference on Programming Language Design and Implementation, PLDI (2006)
9. Hartmann, B., MacDougall, D., Brandt, J., Klemmer, S.R.: What would other programmers do? Suggesting solutions to error messages. In: ACM Conference on Human Factors in Computing Systems, CHI (2010)
10. Hirzel, M., Grimm, R.: Jeannie: Granting Java native interface developers their wishes. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2007)
11. Huang, S.S., Smaragdakis, Y.: Expressive and safe static reflection with MorphJ. In: ACM Conference on Programming Language Design and Implementation, PLDI (2008)
12. Kim, I.-S., Yi, K., Calcagno, C.: A polymorphic modal type system for LISP-like multi-staged languages. In: ACM Symposium on Principles of Programming Languages, POPL (2006)
13. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: ACM Conference on LISP and Functional Programming, LFP (1986)
14. Lerner, B.S., Flower, M., Grossman, D., Chambers, C.: Searching for type-error messages. In: ACM Conference on Programming Language Design and Implementation, PLDI (2007)
15. GNU M4 macro processor, http://www.gnu.org/software/m4/manual/m4.html
16. Mendell, M., Nasgaard, H., Bouillet, E., Hirzel, M., Gedik, B.: Extending a general-purpose streaming system for XML. In: International Conference on Extending Database Technology, EDBT (2012)
17. Moggi, E., Taha, W., Benaissa, Z.-E.-A., Sheard, T.: An Idealized MetaML: Simpler, and More Expressive (Includes Proofs). In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 193–207. Springer, Heidelberg (1999)
18. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
19. Roskind, J.: Parsing C, the last word. The comp.compilers newgroup (January 1992), http://groups.google.com/group/comp.compilers/msg/c0797b5b668605b4
20. Shalit, A.: The Dylan Reference Manual. Addison-Wesley (1996)
21. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. ACM SIGPLAN Notices 37(12) (December 2002)
22. Strachey, C.: A general purpose macrogenerator. The Computer Journal (1965)
23. Stroustrup, B.: The C++ Programming Language. Addison Wesley (2000)
24. Tatlock, Z., Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: Deep typechecking and refactoring. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2008)
25. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: The ASF+SDF compiler. ACM Transactions on Programming Languages and Systems (TOPLAS) 24(4) (July 2002)
26. Weise, D., Crew, R.: Programmable syntax macros. In: ACM Conference on Programming Language Design and Implementation, PLDI (1993)
27. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: ACM Conference on Programming Language Design and Implementation, PLDI (2010)

Practical Permissions for Race-Free Parallelism

Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar

Rice University, Houston, TX 77005, USA
{emw4,jisheng.zhao,zoran,vsarkar}@rice.edu

Abstract. Type systems that prevent data races are a powerful tool for parallel programming, eliminating whole classes of bugs that are both hard to find and hard to fix. Unfortunately, it is difficult to apply previous such type systems to "real" programs, as each of them are designed around a specific synchronization primitive or parallel pattern, such as locks or disjoint heaps; real programs often have to combine multiple synchronization primitives and parallel patterns. In this work, we present a new permissions-based type system, which we demonstrate is practical by showing that it supports multiple patterns (e.g., task parallelism, object isolation, array-based parallelism), and by applying it to a suite of non-trivial parallel programs. Our system also has a number of theoretical advances over previous work on permissions-based type systems, including aliased write permissions and a simpler way to store permissions in objects than previous approaches.

1 Introduction

As computer vendors turn towards multi-core processors for continued performance increases, more and more programmers in the computer industry are faced with the prospect of writing, modifying, testing, and debugging parallel programs. However, working with parallel programs can be challenging due to potential *data races*. Data races can cause programs to run in unexpected and counter-intuitive ways, making parallel programs hard to write, debug, and reason about. While the possible effects of data races have been formalized using complex *memory models* [23,8], just determining whether a race could occur, let alone what the effects of that race could be, is a significant effort. This is true even for parallelism experts working with very small programs.

There has been much research into programming languages and type systems that completely avoid data races [7,33,6,25,1,32,12,11,19,2]. The assumption is that a data race is a bug, which is, in practice, true for most application software. Unfortunately, it is difficult to apply these past approaches to real programs, because past approaches are generally designed around specific synchronization primitives or parallel patterns, such as locks or disjoint heaps. Real programs often have to combine multiple synchronization primitives and parallel patterns, both to get good performance and to match specific algorithms. Even when previous approaches can be used, they often require a high degree of programmer effort, in terms of annotating the code and re-factoring existing parallel

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 614–639, 2012.

algorithms to fit specific parallel patterns. Further, many programmers are not trained to use the sophisticated type systems required by these approaches.

In this paper, we present a *practical* race-free type system that supports multiple patterns (e.g., task parallelism, object isolation, array-based parallelism), and has been applied to a suite of non-trivial parallel programs. Our system is called Habanero Java with permissions (HJp) and is an extension of the Habanero Java (HJ) language [14], which itself is a task-parallel extension of Java. The core idea in HJp is that each object, at each point in time, is in one of two modes: *shared read*, where any task[1] is permitted to read from it but none is permitted to write to it; or *private read-write*, where only one task can read from or write to the object. We enforce this property by extending *permission types* [21,35,13,4,5,12], which capture knowledge about the mode of an object at different points in the program. We introduce two technical advances to permission types. First, our system allows aliased write permissions, while prior work allows write permissions only to non-aliased pointers, which is restrictive in practice.[2] Second, our system introduces a novel approach called *storable permissions* for expressing transitive permissions, e.g., permissions to all elements of a linked list, that requires less technical machinery than previous approaches [13].

To demonstrate the practicality of our approach, we have ported 15 benchmarks from HJ to HJp, totaling almost 14,000 lines of code and covering a range of parallel patterns. This only required modifications to 5% of the lines of code on average. We compared HJp to Deterministic Parallel Java (DPJ) [7,6], another race-free type system: for the same 5 benchmarks, HJp required modifications to 7.3% of the lines of code on average, as opposed to the 10.5% of the code that required annotations in DPJ. Further, HJp is a *gradual* [28] extension of HJ, meaning that some or all of these annotations can be omitted, and the compiler will insert dynamic type-casts where necessary to ensure race-freedom at run time. A simple and effective algorithm for inserting these type-casts was covered in prior work [34], which introduced a runtime approach for checking permissions on entry to regions of code identified by the programmer. The HJp type system presented here expands on that work by introducing programmer annotations that can reduce the number of type-casts inserted, eventually leading to a statically-verified program with no type-casts.

The remainder of this paper is organized as follows. In Sections 2, 3 and 4 we introduce and discuss the main features of the HJp type system: fractional read/write permissions, storable permissions and gradual typing, using a core calculus, called Core HJp, which is then formalized and proved to be race-free in Section 5. Extensions to Core HJp to support array-based parallelism and objects guarded by critical sections are given in Section 6. Practical experience using HJp is then discussed in Section 7, before discussing related work and concluding in the last two sections.

[1] We use "task" instead of "thread" to distinguish potential parallelism in a program from OS threads that may be used to implement this parallelism.

[2] Bierhoff and Aldrich [5] allow aliased writes for protocol enforcement, but it is unclear if their approach applies to race-free parallel programming.

2 Fractional Read/Write Permissions

Fractional permissions, first introduced by Boyland [12], are a powerful idea for fork/join parallel programming. The basic idea is that a task can start out with some permission p, and can temporarily split p in half, yielding two $\frac{1}{2}p$ permissions, both of which can be used in parallel. When parallel use of these permissions are finished, the two $\frac{1}{2}p$ permissions can be re-combined into p again. The exact fractional $\frac{1}{2}$ is not important, just that the two pieces sum to 1. For example, the following code uses permission p to read field f from x:

$$\textbf{finish}\ (\textbf{async}\ (\ldots = x.f);\ \ \ldots = x.f)$$

This code uses HJ's **async** to fork a child task, which is intuitively passed a $\frac{1}{2}p$ permission from the parent to allow it to read $x.f$. The parent continues to run in parallel, using the remaining $\frac{1}{2}p$ permission to allow it to read $x.f$ as well. The **finish** construct then indicates a join on all child tasks spawned inside the lexical block. After this join completes, the parent has p again.

The same pattern, however, cannot be allowed if p is a read/write permission to $x.f$, since this could potentially allow data races. Specifically, it should not be possible for a task to pass a fractional read/write permission to another task while still retaining any read and/or write permissions to the same object. To address this issue, previous permissions-based type systems only allow writes when a task has an *exclusive* (or "unique") permission, i.e., all of the permissions to an object. Exclusivity is indicated by the fraction "1". This guarantees that no other task can read while a task is writing. Unfortunately, it effectively means that writes are only allowed through unique references, which can be very restrictive to the programmer, especially in an OO setting. For example, with standard fractional permissions, calling even the simple function

$$\mathsf{f}\ (x, y) = x.f := y.f;$$

requires that either x and y are statically known to be distinct, or that they are statically known to be aliased so that the permission for x can be re-used for y. Such proofs require complex machinery in the type system and, further, the number of cases grows exponentially with the number of variables.

HJp, in contrast, does allow fractional write permissions. The key idea is this: it is perfectly fine to form fractional write permissions, as long as they *cannot be passed to other tasks*. Thus, HJp has two sorts of fractional permissions: shared read permissions that can be passed to other tasks; and private write permissions that cannot be passed to other tasks. In this way, a program can write to an object without having to show that the pointer is unique. This is especially useful for gradual typing (see Section 4), as it allows code to be instrumented with dynamic acquires and releases of permissions, without having to worry about potential aliases.

In more detail, HJp defines permissions syntactically as follows:

$$\begin{array}{ll} w ::= 0w \mid 1w \mid \varepsilon & \phi ::= w\mathrm{R} \mid w\mathrm{W} \mid w\mathrm{S} \\ \pi ::= x : \phi \mid \mathrm{F}(\pi) & \Pi ::= \Pi, \pi \mid \cdot \end{array}$$

The *permission words* w define the "fractionalness" of a permission. Intuitively, any fractional permission ϕ can be split into two pieces, $0w\phi$ and $1w\phi$, which can be thought of as the left half and the right half of ϕ, respectively. Again, the actual permission word w is not important except to track the splitting and recombining of permissions. The *object permissions* ϕ include read permissions $w\mathrm{R}$, write permissions $w\mathrm{W}$, and sharing permissions $w\mathrm{S}$. We often write Υ to stand for R, W, or S below. Read and write permissions are straightforward, while the sharing permission $w\mathrm{S}$ allows any permissions $w\Upsilon$ with the same permission word w, including the sharing permission itself, to be passed to another task. Note that passing is linear, i.e., permissions cannot be duplicated.

The *permissions* π include: permissions $x : \phi$, that represent permission ϕ to the object pointed to by x; and *future permissions* of the form $\mathrm{F}(\pi)$, which state that the current task will have permission π after the next enclosing **finish** completes. The latter form captures the fact that some child task will hold permission π when it completes. Finally, the *permission sets* Π include zero or more permissions π. We write $\Pi|_l$ to denote the set of all permissions $l : \phi$ in Π.

Permissions are defined this way for theoretical succinctness, but can only occur in practice in permission sets built from one of the following four "building blocks" that represent the programmer view of permissions (see Section 7). Private read permissions $x : w\mathrm{R}$ allow reads $x.f$ of fields of x. Private read/write permissions $x : w\mathrm{R}, x : w\mathrm{W}$ also allow writes to x. Neither of these can be shared with other tasks. Shared read permissions $x : w\mathrm{R}, x : w\mathrm{S}$ allow reads of x and can be shared with other tasks, but do not allow writes. Finally, exclusive permissions $x : \varepsilon\mathrm{R}, x : \varepsilon\mathrm{W}$ (W and S are interchangeable here, by our permission equality rules) allow reads from and writes to $x.f$, and can further be shared with other tasks. Splitting an exclusive permission, though, will disallow either writes or sharing, depending on how it is split. We sometimes abbreviate exclusive permissions as $l : \mathrm{X}$.

Note that, in HJp, there are no shared mutable variables. Local variables are passed by value to child tasks, and global variables are fields in static class objects, as in Java. Thus we do not include permissions for accessing the variable x itself, and instead all permissions $x : \phi$ refer to the object pointed to by x. This also implies an object granularity of permissions, instead of a field granularity. There is no inherent problem in extending HJp to include field permissions, but we have found in practice that this is not needed. Note that HJp does support array-based parallelism, though; see Section 6.

Permission sets are considered equal up to permutations and the rules

$$\Pi, (x : 0w\Upsilon), (x : 1w\Upsilon) = \Pi, x : w\Upsilon \qquad\qquad \Pi, x : \varepsilon\mathrm{W} = \Pi, x : \varepsilon\mathrm{S}$$

The first rule allows the left half $0w\Upsilon$ and the right half $1w\Upsilon$ of a permission $w\Upsilon$ to be combined. Looking at it the other way, the rule allows $w\Upsilon$ to be split into $0w\Upsilon$ and $1w\Upsilon$. The second rule allows a write permission to be exchanged for a sharing permission, or vice-versa, but only when the permission is not fractional, i.e., when exclusive permissions are held. This means that exclusive permissions can either be split into private write or shared read permissions, but not both. Permission set Π_1 is said to *subsume*, or be *more permissive* than, Π_2, written

$\Pi_1 \geq \Pi_2$, iff adding more permissions to Π_2 can yield Π_1; i.e., this holds iff there is some Π_3 such that $\Pi_2, \Pi_3 = \Pi_1$.

3 Storable Permissions

Storable permissions allow permissions to refer to a whole tree of objects instead of just a single object. The idea is that both an object o_2 and exclusive permissions to o_2 can be stored in an object field $o_1.f$. A task with permission ϕ to o_1 can then read o_2 along with ϕ permissions to o_2 out of $o_1.f$, then read ϕ permissions to an object o_3 that is stored in o_2, etc., transitively yielding permission ϕ to a whole group of objects reachable through o_1.

To use storable permissions, some of the fields of class C are designated as *exclusive fields*. (See the **exclusive** keyword in Surface HJp in Section 7.) We write f^{X} for exclusive fields, and write f^{n} for normal, non-exclusive fields. An exclusive field assignment $x.\mathrm{f}^{\mathrm{X}} := y$ implicitly stores an exclusive permission to y in x, meaning that the current task must hold $y : \varepsilon\mathrm{R}, y : \varepsilon\mathrm{W}$ permissions to execute this assignment and these permissions are not held after the assignment.[3] For exclusive field reads $y = x.\mathrm{f}^{\mathrm{X}}$, if the current task holds permission ϕ to x before the read, it then "borrows" this permission from x, yielding a ϕ permission to y after the field read. To specify which object permissions are being borrowed, exclusive field reads are annotated, as $x.\mathrm{f}^{\mathrm{X}}(\vec{\phi})$, where $\vec{\phi}$ indicates a sequence of zero or more object permissions ϕ. The borrowed permissions can then be given back with a remit, written $\mathbf{remit}_y(\vec{\phi} \to x.\mathrm{f}^{\mathrm{X}})$. For example, the code

$$\mathbf{let}\ y = x.\mathrm{f}^{\mathrm{X}}(w\mathrm{R}, w\mathrm{W})\ \mathbf{in}\ y.\mathrm{f}^{\mathrm{n}} := 1; \mathbf{remit}_y(w\mathrm{R}, w\mathrm{W} \to x.\mathrm{f}^{\mathrm{X}})$$

borrows private write permission to $y = x.\mathrm{f}^{\mathrm{X}}$ to write 1 to $y.\mathrm{f}^{\mathrm{n}}$, and then gives back these permissions when complete.

In order to prevent duplication of permissions, the same permissions $\vec{\phi}$ may not be borrowed a second time until either: a remit gives back the borrowed permissions; or another object z is assigned to $x.\mathrm{f}^{\mathrm{X}}$. To track this, we expand the syntax of object permissions as follows:

$$\phi ::= w\mathrm{R} \mid w\mathrm{W} \mid w\mathrm{S} \mid \phi - \mathrm{f}^{\mathrm{X}}@y$$

The new object permission $w\Upsilon - \mathrm{f}^{\mathrm{X}}@y$ indicates that $w\Upsilon$ has been borrowed in variable y for field f^{X}. The construct $-\mathrm{f}^{\mathrm{X}}@y$ is called a *permission subtraction*, and object permissions are considered equal modulo reordering of permission subtractions.[4] The permission subsumption relation is expanded so that $x : \phi \geq x : \phi - \mathrm{f}^{\mathrm{X}}@y$ and so that $x : \phi \geq x : \phi'$ implies $x : \phi - \mathrm{f}^{\mathrm{X}}@y \geq x : \phi' - \mathrm{f}^{\mathrm{X}}@y$. We write $\mathbf{root}(\phi)$ in the below to denote the *root* $w\Upsilon$ of $\phi = w\Upsilon - \vec{B}$.

[3] We could allow shared read permissions to be stored as well, but storing private read or write permissions could allow them to be shared, thereby hurting type soundness.

[4] The permission $x : w\Upsilon - \mathrm{f}^{\mathrm{X}}@y$ can be defined using linear implication as the permission set $x.\neg\mathrm{f}^{\mathrm{X}} : w\Upsilon, (y : w\Upsilon \multimap x.\mathrm{f}^{\mathrm{X}} : w\Upsilon)$ where $x.\mathrm{f}^{\mathrm{X}} : w\Upsilon$ and $x.\neg\mathrm{f}^{\mathrm{X}} : w\Upsilon$ (not defined here) are permissions to just $x.\mathrm{f}^{\mathrm{X}}$ and to all $x.f$ with $f \neq \mathrm{f}^{\mathrm{X}}$, respectively.

$$\begin{aligned}&\mathsf{search} :: (this : \mathsf{List}, \xi \setminus this : \xi\mathrm{R}) \to (\mathsf{Bool} \setminus this : \xi\mathrm{R})\\&\mathsf{search}\ (this, \xi) = \textbf{if}\ P(this.data)\ \textbf{then}\ \mathsf{true}\\&\qquad\qquad\textbf{else if}\ this.next == \textbf{null}\ \textbf{then}\ \mathsf{false}\\&\qquad\qquad\textbf{else let}\ x = this.next(\xi\mathrm{R})\ \textbf{in let}\ r = \mathsf{search}\ (x, \xi)\ \textbf{in}\\&\qquad\qquad\qquad\textbf{remit}_x(\xi\mathrm{R} \to this.next); r\end{aligned}$$

Fig. 1. List Searching with Storable Permissions

As an example, Figure 1 shows how to write a list searching algorithm, search, with storable permissions. We assume a class List that has fields *data* and *next*, containing the data at the head of the list and the next list element, respectively. The *next* field is exclusive, so holding a permission to the head of a list is equivalent to holding the permission for the whole list. The type of search specifies both types and permissions, separated with a backslash, for input and output. The type portion states that search is a function from a List to a Bool. The permissions portion states that, on entry to search, private read permission ξR to the argument *this* is held, and that it will still be held on exit, where ξ is a permission word variable representing an arbitrary permission word.

The search function first tests if some predicate P holds of $this.data$, returning true if so. If not, then it checks $this.next$, returning false if this is **null** and recursing on $this.next$ otherwise. In order to recurse, search first performs an exclusive read of $this.next$, borrowing permission ξR and binding the result to x. The recursive call itself has the form search (x, ξ), which passes x for the argument and ξ for the permission word argument of search. The result of the recursive call is bound to variable r, and the permission ξR on x is given back to $this.next$ before the function returns r.

4 Gradual Permission Types in HJp

Gradual type systems [35,28] are systems which allow some or all of the type-checking of a program to be performed dynamically, rather than statically. This turns strongly typed programming into a gradual process, where the programmer can iteratively add typing annotations to a program to reduce the number of dynamic checks, as opposed to an all-or-nothing effort of passing the type-checker. HJp uses this idea to allow programmers to omit permission annotations where they might not be known by the programmer, or might be hard to guarantee statically. In fact, HJp programs can be compiled and run with absolutely no permission annotations. This means HJp can compile HJ programs that were written with no knowledge of the permissions and HJp still guarantees race-freedom, though some of the dynamic checks may fail at runtime. On the other end of the spectrum, programs with enough permission annotations to not require any dynamic checks are guaranteed not to have any checks that fail at runtime, and, further, suffer no performance loss from dynamic checks. Gradualness in HJp is thus a powerful tool for programmer productivity, making it easy to port existing HJ programs and to write new programs with little or no

initial concern for permissions, while at the same time giving the programmer the ability to gradually perform more work to get more static guarantees.

In previous work [34], we gave a simple but powerful approach to inserting dynamic permission checks. The checks were coarse enough that the runtime overhead was relatively low for un-annotated HJ programs — the slowdown for most benchmarks was under 2.5×, as compared to the order of magnitude slowdown for some of the best dynamic race detectors — but also avoided false positives, i.e., spurious failures of dynamic checks: in 11 HJ benchmarks totaling over 9,000 lines of code, only one modification was needed to prevent false positives.[5] This latter property is especially important for HJp, as it means that the approach generally captures programmer intent, even in HJ code that was not written with permissions in mind. HJp builds upon this work by providing a type system that allows permissions to be fully statically checked. In this section, we briefly review how dynamic permission checks are inserted, and discuss how these checks fit into the type system of HJp.

To support gradual permission typing, Core HJp includes *acquires* and *releases*. An acquire tries to dynamically acquire private read, private write, shared read, or exclusive permissions to x. This succeeds if the requested permission does not conflict with any other permissions held for x. Private reads conflict with private writes in another task, shared reads conflict with any writes, and exclusive conflicts with any other permission. A conflict causes the acquire to fail, and a runtime exception is thrown. A release then gives a permission back to the runtime, indicating that the current task is finished using it.

The fact that acquires throw exceptions on failure mean that HJp implements a fail-stop semantics for data races. This is similar to other work, such as DRFx [29,24], and means that we view data races as bugs, which is in fact the case most of the time. While an alternate approach would be for acquires to block (awaiting the availability of a permission) instead of throwing exceptions, this would add the significant possibility of deadlock, especially since the compiler inserts acquires and releases, as discussed below, that may be unknown to the programmer. Instead, acquires and releases are meant to act as a form of dynamic type cast, and so should change the semantics as little as possible.

As a side note, although HJp prevents data races, it does not guarantee *determinism*. Different acquires could fail for different executions of the same program, depending on the dynamic schedule, or some execution could report no failures at all. If an execution contains no failures, however, then it is guaranteed to contain no races, though it could still contain *potential* races.

Exclusive acquires are written $\mathbf{acquire}_{\mathrm{X}}(x)$. This construct attempts to acquire exclusive permissions for x, returning $*$ on success and raising an exception on failure. Non-exclusive acquires are written $\mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(x)\ \mathbf{in}\ M$, where the sequence $\vec{\Upsilon}$ can be either R for private read, R, W for private read-write, or R, S for shared read permissions. If successful, M is executed with, respectively, permissions $x : w\mathrm{R}$, permissions $x : w\mathrm{R}, x : w\mathrm{W}$, or permissions $x : w\mathrm{R}, x : w\mathrm{S}$. We abbreviate these permissions as $x : w\vec{\Upsilon}$ below. Further, w is also substituted

[5] Array-based parallel loops also had to be modified to use new syntax; see Section 6.

for the permission word variable ξ in M. Note that programs are not allowed to acquire specific permission words w, as this would greatly complicate the implementation of these checks.

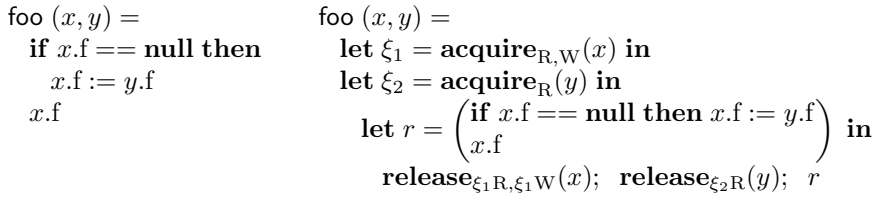
Releases are written $\mathbf{release}_{\vec{\phi}}(x)$. One caveat about releases is that we wish to ensure that a permission can only be released if it was previously acquired. Otherwise, the values of the permission words would matter, complicating the implementation. To do this, we introduce a new permission form $\mathrm{dyn}_x(w\vec{\Upsilon})$, called a *dynamic permission*, which indicates that the sequence $\vec{\Upsilon}$ of object permissions were acquired dynamically with permission word w. A release $\mathbf{release}_{w\vec{\Upsilon}}(x)$ then consumes both the permissions $x : w\vec{\Upsilon}$ and the permission $\mathrm{dyn}_x(w\vec{\Upsilon})$. Exclusive permissions do not involve a permission word w, and so are a special case that do not require a dynamic permission to be released.

In order to support gradual typing, the HJp compiler automatically inserts acquires and releases where necessary. The basic idea, as presented in previous work [34], is to add acquires and releases around variable scopes for the least permission — private write, private read, or none — needed for the given variable. This essentially creates regions of code that are as large as possible during which the current task holds permissions to the given object. As we prove in Section 5, any insertion algorithm will prevent low-level data races (as defined by the Java Memory Model [23]). Making the regions as large as possible, however, also helps to prevent high-level races, where an object is modified concurrently (without a low-level race) while the programmer intends for it to remain constant. Of course, it is impossible in general to infer programmer intent, but the HJp insertion algorithm seems to seems to capture a "sweet spot" with the intuition that programmers do not generally intend objects to be modified while they are in scope as variables. In addition, our experiments on HJ code, which were written without permissions in mind, also showed that in only one instance for all our benchmarks were these regions too big.

Note that, as a special case, object constructors are implicitly considered to be exclusive acquires, meaning that *new* returns exclusive permissions to an object. Thus the HJp insertion algorithm inserts corresponding releases at the end of an allocated object's scope.

Figure 2 illustrates the HJp insertion algorithm, also demonstrating the use of acquires and releases. The figure shows a simple function, $\mathsf{foo}(x, y)$, which first checks if x.f is **null**, assigning y.f to it if so, and then returns the possibly modified value of x.f. Figure 2(b) demonstrates how acquires and releases are added to this code: a permission region is inserted around the body of the entire function, since this is the scope of the variables x and y. The resulting code starts by acquiring write permission to x and (private) read permission to y. It then performs the original computation, binding the result to a new variable r. Finally, the code releases permissions to x and y and returns r.

Note that a more "conservative" approach would only acquire private read permission to x outside the if-expression, acquiring write permission to x and private read permission to y only when x.f equals **null**. The HJp insertion algorithm, however, captures the fact that, *logically*, calling $\mathsf{foo}(x, y)$ requires read



(a) Original Code (b) After Compiler Insertion

Fig. 2. Compiler Insertion of Acquires and Releases

permission to y and write permission to x. A violation of this logic, such as a concurrent write to (the object referenced by) y, is therefore considered to be a data race in HJp, even if the read from y does not actually take place. The user can override this behavior by manually inserting acquires and releases. The user can also completely remove any acquires and releases by changing the type of foo, to indicate that write permission to x and read permission to y are required when foo is called. This is denoted in Surface HJp by adding keywords **writing** and **reading** to x and y, respectively (see Section 7). We explain how this type is written in Core HJp in Section 5. Modifying the type of foo in this way represents a gradual refinement, moving the function foo from dynamic to static type-checking. Since the resulting function has no dynamic permission checks, it is statically guaranteed not to have any data races, though races in callers of foo could lead to exceptions being thrown.

As a final point here, the code inserted by the HJp compiler for acquires and releases is actually slightly more complex than that shown in Figure 2, for two reasons. First, the code must handle the case where an exception is thrown between an acquire and a release. This is done with a try-finally block to ensure that releases are always performed at the end of permission regions. The second reason that the inserted code is more complex is to handle mutable variables in Java. A permission region for a mutable variable x causes the specified permission to be re-acquired for the new value of x each time x is modified. All of the permissions acquired for values of x are not released until the end of the permission region, when all of the permissions are released. Both of these points are discussed more in our previous work [34], and can be modeled in a straightforward way in Core HJp.

5 Syntax and Semantics of Core HJp

In this section, we formalize Core HJp and prove that all executions are race-free. Core HJp is defined using A-normal forms, similar to 3-address codes, where a term consists of a sequence of steps, each of which returns a value that can be **let**-bound in the remainder of the computation. This closely matches the Jimple representation of the Soot optimization framework [31], which is used as a back-end for our HJp implementation. The typing judgment associates input, output, and exception permissions with each term M, in a manner similar to Hoare Type

Theory [26], where the typing judgment associates pre- and post-conditions with terms. We omit a number of high-level features present in Java, HJ and Surface HJp, such as subtyping, constructors, final fields, methods, and conditionals; objects in Core HJp are essentially nominally typed, mutable records. The other features are straightforward to add, but do not significantly affect the results.

The remainder of this section is organized as follows. Section 5.1 gives the syntax and type system for HJp. Section 5.2 then gives an operational semantics for HJp, and proves the main result, that any execution of HJp is race-free. Note that, although our operational semantics is a sequentially consistent semantics, proving that all executions are race-free means that all HJp programs are *correctly synchronized*. This in turn ensures that any execution under a weak, DRF0-based memory model (such as the Java Memory Model) is equivalent to a sequentially consistent execution as given here [23].

5.1 Static Semantics

To define the syntax of Core HJp, we first fix sets of class names and fields, written C and f, respectively, as well as countably infinite sets of term variables, written x, y, and z, and word variables, written ξ. We also syntactically distinguish the normal fields f^{n} from the fields f^{X} marked as exclusive. To denote sequences of zero or more syntactic elements, we use an arrow over a syntactic category. For example, $\vec{x}$ indicates a sequence of zero or more variables, referred to as x_1 through x_n. Further, we often use this notation as a shorthand in compound notation; e.g., $\vec{\mathrm{f}^{\mathrm{n}}} : \vec{\tau}$ denotes a sequence $\mathrm{f}_1^{\mathrm{n}} : \tau_1, \ldots, \mathrm{f}_n^{\mathrm{n}} : \tau_n$, while $x : \vec{\phi}$ denotes a sequence of permissions $x : \phi_1, \ldots, x : \phi_n$.

The syntax of Core HJp is given in Figure 3. This includes the permissions and permission sets as defined in Section 2 along with the permission subtractions $-\mathrm{f}^{\mathrm{X}}@y$ defined in Section 3 and the permission word variables ξ and dynamic permissions $\mathrm{dyn}_x(\vec{\phi})$ described in Section 4. The Core HJp types τ include the class names C as well as the types $(\Gamma \setminus \Pi_{\mathrm{i}}) \to (y : \tau \setminus \Pi_{\mathrm{o}} \setminus \Pi_{\mathrm{e}})$ of functions that take, as input, values and permission words whose types are specified by Γ. In practice, Γ always has the form $x : \tau_x, \vec{\xi}$, for functions that take one input but quantify over zero or more permission words. The output type is given by τ. The permission set Π_{i} specifies the minimal (under subsumption) permissions that must be held to call the function, while Π_{o} and Π_{e} specify the permissions returned under normal and exceptional exit from the function. The variables x and $\vec{\xi}$ are considered bound in the remaining constructs, while the variable y, which refers to the value of the output, is considered bound in Π_{o} and Π_{e}. The latter allows Π_{o} and Π_{e} to refer to the output of the function.

Next in Figure 3 are the signatures, written Σ, and the type contexts, written Γ or Δ. A signature associates each class name C in a program with a list of fields and their associated types. We assume a fixed signature for the remainder of this section, which we write as Σ below. A typing context Γ associates types with term variables x, and lists word variables ξ that are considered to be in scope; any variables listed in context Γ are considered bound in later types. We implicitly assume that all signatures and typing contexts are well-formed,

$$
\begin{array}{ll}
w ::= 0w \mid 1w \mid \varepsilon \mid \xi & \Upsilon ::= \mathrm{R} \mid \mathrm{W} \mid \mathrm{S} \\
\phi ::= w\Upsilon \mid \phi - \mathrm{f}^{\mathrm{X}}@y & \pi ::= x : \phi \mid \mathrm{F}(\pi) \mid \mathrm{dyn}_x(\phi) \\
\Pi ::= \Pi, \pi \mid \cdot & \tau ::= C \mid (\Gamma \setminus \Pi_{\mathrm{i}}) \to (y : \tau' \setminus \Pi_{\mathrm{o}} \setminus \Pi_{\mathrm{e}}) \\
\Sigma ::= \cdot \mid \Sigma, C \mapsto \{\vec{\mathrm{f}^{\mathrm{n}}} : \vec{\tau}, \vec{\mathrm{f}^{\mathrm{X}}} : \vec{C}\} & \Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \xi
\end{array}
$$

$$
\begin{array}{l}
M ::= x \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \mid x\ (y, \vec{\xi}) \mid \lambda(x, \vec{\xi}).M \mid \mathbf{exn} \mid \mathbf{null} \mid x.\mathrm{f}^{\mathrm{n}} \mid x.\mathrm{f}^{\mathrm{n}} := y \\
\quad \mid x.\mathrm{f}^{\mathrm{X}}(\vec{\phi}) \mid x.\mathrm{f}^{\mathrm{X}} := y \mid \mathbf{remit}_x(\phi \to y.\mathrm{f}^{\mathrm{X}}) \mid \mathbf{new}\ C\ \langle \vec{\mathrm{f}} \mapsto \vec{x} \rangle \mid \mathbf{acquire}_{\mathrm{X}}(x) \\
\quad \mid \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(x)\ \mathbf{in}\ M \mid \mathbf{release}_{\vec{\phi}}(x) \mid \mathbf{async}_{\Pi}\ M \mid \mathbf{finish}\ M
\end{array}
$$

Fig. 3. Syntax

meaning they contain only bound occurrences of variables x and ξ. In the below, we use the notations $\Gamma(x)$, $\Sigma(C)$, and $\Sigma(C)(\mathrm{f})$ to denote, respectively, the type associated with x in Γ, the sequence of field-type pairs associated with C in Σ, and the type in this sequence associated with f.

The terms of Core HJp are in A-normal form, in order to allow permission sets to refer to each intermediate value as a variable. Thus field reads and writes, acquires, and releases all refer only to variables as their arguments. Compound expressions can be built using **let**-expressions. Functions have the form $\lambda(x, \vec{\xi}).M$, meaning that they quantify over a value x and zero or more permission words $\vec{\xi}$. Similarly, function applications have the form $x\ (y, \vec{w})$, applying variable x to argument y and permission words $\vec{w}$.

The typing judgment $\Gamma \setminus \Pi_{\mathrm{i}} \vdash M : y : \tau \setminus \Pi_{\mathrm{o}} \setminus \Pi_{\mathrm{e}}$ for Core HJp is defined in Figure 4. It states that M is well-typed under typing context Γ and assuming input permissions Π_{i}, and M returns a value of type τ and permissions given either by Π_{o} or Π_{e}, depending on whether it had a normal or exceptional exit, respectively. We sometimes omit the exceptional permission set Π_{e} when it is allowed to have any value, which implies that the given expression cannot throw an exception. Note that we implicitly assume, for each rule, that the type τ and the permission sets Π_{o} and Π_{e} are well-formed, meaning they contain only variables listed in Γ and, in the case of Π_{o} and Π_{e}, possibly y.

The first rule, T-Sub, allows permissions to be added to the input permissions Π_{i} or to be removed from the output permissions Π_{o} and Π_{e}. It also allows the same permissions to be added to both sides, in a manner similar to the frame rule of separation logic. T-Var types variables x, allowing the output permissions to refer to the output itself as a variable, y, instead of to the variable x; output permissions that still refer to x can be added with T-Sub. Since variables cannot throw exceptions, any Π_{e} may be used. T-Let types **let** $x = M_1$ **in** M_2 by binding x in the type context for M_2, and also states that the output permissions for M_1 must serve as input permissions to M_2. Both must have the same exception permission Π_{e}, since an exception from the **let**-expression could be thrown from either M_1 or M_2.

T-App types applications $x\ (y, \vec{w})$, turning the function type for x into a typing assertion by substituting the arguments y and $\vec{w}$ into the type for x. Functions $\lambda(x, \vec{\xi}).M$ do the opposite. Note that functions do not consume or produce any permissions, since their bodies do not run until they are called.

$$\frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e} \qquad \Pi_\mathrm{o} \geq \Pi'_\mathrm{o} \qquad \Pi_\mathrm{e} \geq \Pi'_\mathrm{e} \qquad \Pi'_\mathrm{i} \geq \Pi_\mathrm{i}}{\Gamma \setminus \Pi'_\mathrm{i}, \Pi' \vdash M : y : \tau \setminus \Pi'_\mathrm{o}, \Pi' \setminus \Pi'_\mathrm{e}, \Pi'} \text{ T-Sub} \qquad \frac{x : \tau \in \Gamma}{\Gamma \setminus \Pi \vdash x : y : \tau \setminus [y/x]\Pi} \text{ T-Var}$$

$$\frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M_1 : x : \tau' \setminus \Pi' \setminus \Pi_\mathrm{e} \quad \Gamma, x : \tau' \setminus \Pi' \vdash M_2 : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}} \text{ T-Let}$$

$$\frac{x : ((y' : \tau, \vec{\xi}) \setminus \Pi_\mathrm{i}) \to (z : \tau' \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}) \in \Gamma \quad \sigma = [y/y', \vec{w}/\vec{\xi}]}{\Gamma \setminus \sigma\Pi_\mathrm{i} \vdash x\ (y, \vec{w}) : z : \tau' \setminus \sigma\Pi_\mathrm{o} \setminus \sigma\Pi_\mathrm{e}} \text{ T-App}$$

$$\frac{\Gamma, x : \tau, \vec{\xi} \setminus \Pi_\mathrm{i} \vdash M : y : \tau' \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}}{\Gamma \setminus \cdot \vdash \lambda(x, \vec{\xi}).M : (x : \tau, \vec{\xi} \setminus \Pi_\mathrm{i}) \to (y : \tau' \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}) \setminus \cdot} \text{ T-Lam} \qquad \frac{}{\Gamma \setminus \cdot \vdash \mathbf{exn} : y : \tau \setminus \Pi_\mathrm{o} \setminus \cdot} \text{ T-Exn}$$

$$\frac{}{\Gamma \setminus \cdot \vdash \mathbf{null} : y : C \setminus y : \vec{\phi}} \text{ T-Null} \qquad \frac{x : C \in \Gamma \quad \mathbf{root}(\phi) = w\mathrm{R}}{\Gamma \setminus x : \phi \vdash x.\mathrm{f^n} : \Sigma(C)(\mathrm{f^n}) \setminus x : \phi \setminus x : \phi} \text{ T-Read}$$

$$\frac{x : C, y : \Sigma(C)(\mathrm{f^n}) \in \Gamma \quad \mathbf{root}(\vec{\phi}) = w\mathrm{R}, w\mathrm{W}}{\Gamma \setminus x : \vec{\phi} \vdash (x.\mathrm{f^n} := y) : \mathbf{U} \setminus x : \vec{\phi} \setminus x : \vec{\phi}} \text{ T-Write}$$

$$\frac{\vec{\phi} = w\vec{\Upsilon} - \vec{B} \quad \mathrm{R} \in \vec{\Upsilon} \quad x : C \in \Gamma \quad \nexists z.\mathrm{f^X}@z \in \vec{B}}{\Gamma \setminus x : \vec{\phi} \vdash x.\mathrm{f^X}(w\vec{\Upsilon}) : y : \Sigma(C)(\mathrm{f^X}) \setminus x : \vec{\phi} - \mathrm{f^X}@y, y : w\vec{\Upsilon} \setminus x : \vec{\phi}} \text{ T-XRead}$$

$$\frac{x : C \in \Gamma \quad y : \Sigma(C)(\mathrm{f^X}) \in \Gamma \quad \mathbf{root}(\vec{\phi}) = w\mathrm{R}, w\mathrm{W}}{\Gamma \setminus x : \vec{\phi}, y : \mathrm{X} \vdash x.\mathrm{f^X} := y : \mathbf{U} \setminus x : \vec{\phi} + \mathrm{f^X} \setminus x : \vec{\phi}, y : \mathrm{X}} \text{ T-XWrite}$$

$$\frac{x : C \in \Gamma \quad y : C' \in \Gamma \quad \mathbf{root}(\phi) = w\Upsilon}{\Gamma \setminus x : \phi, y : w\Upsilon \vdash \mathbf{remit}_y(w\Upsilon \to x.\mathrm{f^X}) : \cdot \setminus x : \phi + \mathrm{f^X}@y} \text{ T-Remit}$$

$$\frac{\Sigma(C) = \{\vec{\mathrm{f^n}} : \Gamma(\vec{x}), \vec{\mathrm{f^X}} : \Gamma(\vec{y})\}}{\Gamma \setminus \vec{y} : \mathrm{X} \vdash \mathbf{new}\ C\ \langle \vec{\mathrm{f^n}} \mapsto \vec{x}, \vec{\mathrm{f^X}} \mapsto \vec{y} \rangle : x : C \setminus x : \mathrm{X}} \text{ T-New}$$

$$\frac{x : C \in \Gamma \quad \Gamma, \xi \setminus \Pi_\mathrm{i}, x : \xi\vec{\Upsilon}, \mathrm{dyn}_x(\xi\vec{\Upsilon}) \vdash M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e} \quad \Pi_\mathrm{i} \geq \Pi_\mathrm{e}}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(x)\ \mathbf{in}\ M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}} \text{ T-Acq}$$

$$\frac{x : C \in \Gamma}{\Gamma \setminus \cdot \vdash \mathbf{acquire}_\mathrm{X}(x) : \mathbf{U} \setminus x : \mathrm{X} \setminus \cdot} \text{ T-AcqX} \qquad \frac{x : C \in \Gamma \quad \forall i.\phi_i = w_i\Upsilon_i}{\Gamma \setminus x : \vec{\phi}, \mathrm{dyn}_x(\vec{\phi}) \vdash \mathbf{release}_{\vec{\phi}}(x) : \mathbf{U} \setminus \cdot} \text{ T-Rel}$$

$$\frac{x : C \in \Gamma}{\Gamma \setminus x : \mathrm{X} \vdash \mathbf{release}_\mathrm{X}(x) : \mathbf{U} \setminus \cdot} \text{ T-RelX} \qquad \frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{o} \quad \mathbf{sharable}(\Pi_\mathrm{i})}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{async}_{\Pi_\mathrm{i}}\ M : \mathbf{U} \setminus \mathrm{F}(\Pi_\mathrm{o})} \text{ T-Async}$$

$$\frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e} \quad \nexists \mathrm{F}(\pi) \in \Pi_\mathrm{i}}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{finish}\ M : y : \tau \setminus \Pi_\mathrm{o} - \mathbf{F} \setminus \Pi_\mathrm{e} - \mathbf{F}} \text{ T-Finish}$$

Fig. 4. Static Semantics

Exceptions **exn** can have any type and output permissions, as they will not return, but they have the same exception permission set as input permission set. T-Exn gives these both as empty permission sets $\cdot$, but permissions can be added to both sides using T-Sub. Similarly, **null** can have any type and any output permissions to the **null** value, since, intuitively, permissions to **null** do not matter; this is made more formal in the operational semantics, below.

Normal field reads and writes require read and write permission, respectively, for the object being accessed. Writes return the sole element $*$ of the singleton

type **U**. Exclusive field reads $x.\mathrm{f}^{\mathrm{X}}(w\vec{\Upsilon})$ require at least read permission to x but without the borrowing permission subtraction $-\mathrm{f}^{\mathrm{X}}@y$, and append this subtraction to the permissions on x. Note that we use $x : w\vec{\Upsilon}$ to denote the permission set $x : w\Upsilon_1, \ldots, x : w\Upsilon_n$. Exclusive writes $x.\mathrm{f}^{\mathrm{X}} := y$ require exclusive permission $y : \mathrm{X}$ (recall that this is shorthand for $y : \varepsilon\mathrm{R}, y : \varepsilon\mathrm{W}$) and erase any subtractions $\mathrm{f}^{\mathrm{X}}@z$ on permissions for x, where $x : \vec{\phi} + \mathrm{f}^{\mathrm{X}}$ indicates the removal of any $\mathrm{f}^{\mathrm{X}}@z$ in each ϕ_i. These ϕ_i must comprise at least a write permission to x. Note that all of these return the input permission set as the exceptional permission set, in the case the the object being read or written is **null**.

The $\mathbf{remit}_y(w\Upsilon \to x.\mathrm{f}^{\mathrm{X}})$ construct erases a subtraction $-\mathrm{f}^{\mathrm{X}}@y$, using the notation $\phi + \mathrm{f}^{\mathrm{X}}@y$, as well as erasing a $w\Upsilon$ permission to y. Object allocation requires X permissions to all objects assigned to exclusive fields and returns an X permission to the newly allocated object.

An exclusive acquire returns an X permission to x as output permission but returns an empty exception permission. Again, using T-Sub allows the input permission set to be anything and requires the exception permission set to be the same (in case the acquire throws an exception) but adds $x : \mathrm{X}$ to the output permission set. A non-exclusive acquire **let** ξ = $\mathbf{acquire}_{\vec{\Upsilon}}(x)$ **in** M provides permissions $\xi\vec{\Upsilon}$ to x in the body M, also indicating that these permissions are dynamic. A $\mathbf{release}_{\vec{\phi}}(x)$ releases the permissions $\vec{\phi}$ for x, requiring that either $\vec{\phi}$ equals exclusive permissions or that there is a dynamic permission $\mathrm{dyn}_x(\vec{\phi})$ to indicate that $\vec{\phi}$ can be dynamically released. An **async** $\Pi_{\mathrm{i}} M$ passes to M the permissions Π_{i}, which must satisfy $\mathbf{sharable}(\Pi_{\mathrm{i}})$. The latter indicates that Π_{i} contains only pairs $x : w\mathrm{R}, x : w\mathrm{S}$ and permissions $\mathrm{dyn}_x(\vec{\phi})$. Any permissions Π_{o} returned by M are returned as future permissions to the current task, where $\mathrm{F}(\Pi)$ maps each π in Π to $\mathrm{F}(\pi)$. Note that an **async** catches any exceptions but does not throw any, so it can have any exceptional permission set, while it requires the output and exception permissions of M to be the same. The latter can always be achieved by T-Sub. Future permissions can be reclaimed with **finish** M, which returns the same permissions as M with any future permissions $\mathrm{F}(\pi)$ turned into π. This is written as $\Pi - \mathbf{F}$, which also converts $\mathrm{F}(\mathrm{F}(\pi))$, etc., to π, to handle nested **async**s. Note that no future permissions from outside the **finish** are passed to M, as these are only available to the next enclosing **finish**.

5.2 Operational Semantics

The additional syntax needed to define the operational semantics is given in Figure 5. Two important changes are that values are now allowed to occur in place of variables in both permissions and terms, and that private read permissions are now annotated with task ids t, where we assume a countably infinite set of task ids. The latter change was unnecessary in the static semantics, since an expression can only refer to permissions for the current task, but will be necessary to model the permissions held globally for an object. There are now distinct private read as well as private read-write permission sets for each task id t.

$$
\begin{array}{lll}
\Upsilon & ::= \mathrm{R}^t \mid \mathrm{W} \mid \mathrm{S} & M ::= \ldots \mid l^C \mid [\vec{v}/\vec{x}]M \mid \mathbf{finish}(M \parallel \vec{T}) \\
\pi & ::= \ldots \mid l : \phi \mid \mathrm{dyn}_l(\phi) & H ::= \cdot \mid H, l \mapsto \langle \mathbf{p} \mapsto \Pi, \vec{f} \mapsto \vec{v} \rangle \\
T & ::= (\Pi \setminus M)^t & E^* ::= \Box \mid \mathbf{let}\ x = E^*\ \mathbf{in}\ M \\
v & ::= \mathbf{null} \mid l^C \mid \lambda(x, \vec{\xi}).M & E ::= \Box \mid E^*[E] \mid \mathbf{finish}(E \parallel \vec{T}) \\
res & ::= v \mid \mathbf{exn} &
\end{array}
$$

Fig. 5. Operational Syntax

We write Π^t for the result of adding task id t to the (un-annotated) private read permissions in Π. These two changes yield the additional equalities:

$$x : w\mathrm{R}^{t_1}, x : w\mathrm{S} = x : w\mathrm{R}^{t_2}, x : w\mathrm{S} \qquad \mathbf{null} : \phi = \cdot$$

The first captures the fact that shared read and exclusive permissions are insensitive to which task holds them, while the second captures the fact that permissions to **null** can be added or removed at will.

The values v include functions, **null**, and heap locations l, where the latter is annotated with its class as l^C. The new term construct $\mathbf{finish}(M \parallel \vec{T})$, which is considered equal modulo permutation of $\vec{T}$, represents a **finish** waiting for parallel tasks $\vec{T}$ to complete. Tasks are written $(\Pi \setminus M)^t$, indicating a task that is executing term M, holds permissions Π, and has task id t. Typing is extended to tasks with the judgement $\vdash T : \Pi_\mathrm{o}$, which holds for task $(\Pi \setminus M)^t$ if and only if $\cdot \setminus \Pi \vdash M : \mathbf{U} \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{o}$. Typing is then extended to $\mathbf{finish}(M \parallel \vec{T})$ by requiring $\vdash T_i : \Pi_i$ for each T_i, and adding $\vec{\Pi}$ to the output and exceptional permission sets returned by the construct.

The results res include exceptions and values, while the heaps H map locations l to heap forms $\langle \mathbf{p} \mapsto \Pi, \vec{f} \mapsto \vec{v} \rangle$. The latter are themselves mappings from fields to values and from the special marker $\mathbf{p}$ to the set of permissions Π of permissions to l still available for dynamic acquires, where Π must contain only permissions $l : \phi$. (We use permission sets here to take advantage of permission set equality.) Thus, e.g., $H(l)(f)$ returns the value of field f at location l in H, while $H(l)(\mathbf{p})$ returns permissions to l that are available for dynamic acquires. As a convenience, each heap H also contains a mapping $H(\mathbf{null})(\mathbf{p}) = \cdot$ which (by the above) equals $\mathbf{null} : \vec{\phi}$ for any $\vec{\phi}$.

Finally, Figure 5 defines the evaluation contexts E, which intuitively define a term with single a "hole" $\Box$ indicating where evaluation can take place in a term. This includes the term being bound in a **let** and the body of a **finish**. We write $E[M]$ for the (non-capture-avoiding) replacement of $\Box$ by M. We also define evaluation contexts E^* out of which exceptions can be thrown. These exclude **finish**, requiring child tasks to complete before an exception leaves a **finish**.

The operational semantics is defined in Figure 6 as a small-step relation $H \setminus \Pi \setminus M \xrightarrow{a} H' \setminus \Pi' \setminus M'$. This judgment states that term M with heap H, in a task holding permissions Π, evaluates in one step to term M', changing the heap to H' and the permissions held by the current task to Π'. This step is also labeled with an action label a, which can have any of the forms given in Figure 6. Action labels can optionally have the prefix $(t; \Pi) : a$, indicating that the action occurred in task id t and yielded permission set Π in task t. Only the inner-most

$$\frac{H \setminus \Pi \setminus M \xrightarrow{a} H' \setminus \Pi' \setminus M'}{H \setminus (\Pi \setminus M)^t \xrightarrow{(t;\Pi'):a} H' \setminus (\Pi' \setminus M')^t}\ \text{E-Task} \qquad \frac{H \setminus \Pi \setminus M \xrightarrow{a} H' \setminus \Pi' \setminus M'}{H \setminus \Pi \setminus E[M] \xrightarrow{a} H' \setminus \Pi' \setminus E[M']}\ \text{E-Ctx}$$

$$\frac{}{H \setminus \Pi \setminus E^*[\mathbf{exn}] \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-Exn} \qquad \frac{}{H \setminus \Pi \setminus \mathbf{let}\ x = v\ \mathbf{in}\ M \xrightarrow{\cdot} H \setminus \Pi \setminus [v/x]M}\ \text{E-Let}$$

$$\frac{}{H \setminus \Pi \setminus (\lambda(x, \vec{\xi}).M)\ (v, \vec{w}) \xrightarrow{\cdot} H \setminus \Pi \setminus [v/x, \vec{w}/\vec{\xi}]M}\ \text{E-App}$$

$$\frac{}{H \setminus \Pi \setminus l.\mathrm{f}^{\mathrm{n}} \xrightarrow{l.\mathrm{f}^{\mathrm{n}} \rightharpoonup H(l)(\mathrm{f}^{\mathrm{n}})} H \setminus \Pi \setminus H(l)(\mathrm{f}^{\mathrm{n}})}\ \text{E-Read} \qquad \frac{}{H \setminus \Pi \setminus \mathbf{null}.f \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-NullR}$$

$$\frac{}{H \setminus \Pi \setminus l.\mathrm{f}^{\mathrm{n}} := v \xrightarrow{l.\mathrm{f}^{\mathrm{n}} \leftharpoonup v} H[(l, \mathrm{f}^{\mathrm{n}}) \mapsto v] \setminus \Pi \setminus *}\ \text{E-Write} \qquad \frac{}{H \setminus \Pi \setminus \mathbf{null}.f := v \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-NullW}$$

$$\frac{\vec{\phi} = w\vec{\Upsilon} - \vec{B} \quad l' = H(l)(\mathrm{f}^{\mathrm{X}})}{H \setminus \Pi, l : \vec{\phi} \setminus l.\mathrm{f}^{\mathrm{X}}(w\vec{\Upsilon}) \xrightarrow{l.\mathrm{f}^{\mathrm{X}} \rightharpoonup l'} H \setminus \Pi, l : \vec{\phi} - \mathrm{f}^{\mathrm{X}}@l', l' : w\vec{\Upsilon} \setminus l'}\ \text{E-XRead}$$

$$\frac{\nexists(l : \phi) \in \Pi}{H \setminus \Pi, l : \vec{\phi}, l' : \mathrm{X} \setminus l.\mathrm{f}^{\mathrm{X}} := l' \xrightarrow{l.\mathrm{f}^{\mathrm{X}} \leftharpoonup l'} H[(l, \mathrm{f}^{\mathrm{X}}) \mapsto l'] \setminus \Pi, l : \vec{\phi} + \mathrm{f}^{\mathrm{X}} \setminus *}\ \text{E-XWrite}$$

$$\frac{\phi = w\Upsilon - \vec{B}}{H \setminus \Pi, v : \phi, v' : w\Upsilon \setminus \mathbf{remit}_{v'}(w\Upsilon \to v.\mathrm{f}^{\mathrm{X}}) \xrightarrow{\cdot} H \setminus \Pi, v : \phi + \mathrm{f}^{\mathrm{X}}@v' \setminus \cdot}\ \text{E-Remit}$$

$$\frac{l \text{ fresh}}{H \setminus \Pi, \vec{v} : \mathrm{X} \setminus \mathbf{new}\ C\ \langle \vec{\mathrm{f}^{\mathrm{X}}} \mapsto \vec{v}, \vec{\mathrm{f}^{\mathrm{n}}} \mapsto \vec{v'} \rangle \xrightarrow{\mathbf{new}(l; \vec{f} \mapsto \vec{v})} H, l \mapsto \langle \mathbf{p} \mapsto \cdot, \vec{\mathrm{f}^{\mathrm{X}}} \mapsto \vec{v}, \vec{\mathrm{f}^{\mathrm{n}}} \mapsto \vec{v'} \rangle \setminus \Pi, l : \mathrm{X} \setminus l}\ \text{E-New}$$

$$\frac{H(l)(\mathbf{p}) \neq l : \mathrm{X}}{H \setminus \Pi \setminus \mathbf{acquire}_{\mathrm{X}}(l) \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-AcqXFail}$$

$$\frac{\nexists(\Pi_l, w).\Pi, H(l)(\mathbf{p}) = \Pi, \Pi_l, l : w\vec{\Upsilon}}{H \setminus \Pi \setminus \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(l)\ \mathbf{in}\ M \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-AcqFail}$$

$$\frac{H(v)(\mathbf{p}) = v : \mathrm{X}}{H \setminus \Pi \setminus \mathbf{acquire}_{\mathrm{X}}(v) \xrightarrow{\mathbf{acq}(v:\mathrm{X})} H[(v, \mathbf{p}) \mapsto \cdot] \setminus \Pi, v : \mathrm{X} \setminus *}\ \text{E-AcqX}$$

$$\frac{\Pi, H(v)(\mathbf{p}) = \Pi, \Pi_v, v : w\vec{\Upsilon} \quad \exists w.\Pi_v \geq w\mathrm{R}^t}{H \setminus \Pi \setminus \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(v)\ \mathbf{in}\ M \xrightarrow{\mathbf{acq}(v:w\vec{\Upsilon})} H[(v, \mathbf{p}) \mapsto \Pi_v] \setminus \Pi, v : w\vec{\Upsilon}, \mathrm{dyn}_v(w\vec{\Upsilon}) \setminus [w/\xi]M}\ \text{E-Acq}$$

$$\frac{\text{if } \vec{\phi} = \varepsilon\mathrm{R}^t, \varepsilon\mathrm{W} \text{ then } \Pi_1 = \cdot \text{ else } \Pi_1 = \mathrm{dyn}_v(\vec{\phi})}{H \setminus \Pi, v : \vec{\phi}, \Pi_1 \setminus \mathbf{release}_{\vec{\phi}}(v) \xrightarrow{\mathbf{rel}(v:\vec{\phi})} H[(v, \mathbf{p}) \mapsto H(v)(\mathbf{p}), v : \vec{\phi}] \setminus \Pi \setminus *}\ \text{E-Rel}$$

$$\frac{t \text{ fresh}}{H \setminus \Pi, \Pi' \setminus \mathbf{finish}(E^*[\mathbf{async}_{\Pi}\ M] \parallel \vec{T}) \xrightarrow{\mathbf{async}(t)} H \setminus \Pi' \setminus \mathbf{finish}(E^*[*] \parallel \vec{T}, (\Pi \setminus M)^t)}\ \text{E-Async}$$

$$\frac{t' \text{ fresh}}{H \setminus \Pi'' \setminus \mathbf{finish}(M'' \parallel \vec{T}, (\Pi, \Pi' \setminus E^*[\mathbf{async}_{\Pi}\ M])^t) \xrightarrow{(t;\Pi'):\mathbf{async}(t')} H \setminus \Pi'' \setminus \mathbf{finish}(M'' \parallel \vec{T}, (\Pi' \setminus E^*[*])^t, (\Pi \setminus M)^{t'})}\ \text{E-AsyncPar}$$

$$\frac{H \setminus T \xrightarrow{a} H' \setminus T'}{H \setminus \Pi \setminus \mathbf{finish}(M \parallel T, \vec{T}) \xrightarrow{a} H' \setminus \Pi \setminus \mathbf{finish}(M \parallel T', \vec{T})}\ \text{E-Par}$$

$$\frac{}{H \setminus \Pi \setminus \mathbf{finish}(res \parallel (\vec{\Pi} \setminus r\vec{es})^{\vec{t}}) \xrightarrow{\mathbf{finish}(\vec{t})} H \setminus \Pi, \vec{\Pi} \setminus res}\ \text{E-Finish}$$

Fig. 6. Operational Semantics

prefix is used, so $(t_1; \Pi_1) : (t_2; \Pi_2) : a$ is considered equal to $(t_2; \Pi_2) : a$. These labels are used below to define the happens-before ordering and data races.

The E-Task rule defines evaluation on tasks, which evaluates the term and permission set in the task and then adds the task id and resulting permission set to a. E-Ctx allows evaluation inside evaluation contexts E, while E-Exn allows exceptions to be thrown out of exception evaluation contexts E^*. E-Let and E-App evaluate **let**-expressions and function applications using substitution.

Normal field reads of $l.\mathrm{f^n}$ return the value $H(l)(\mathrm{f^n})$ associated with $\mathrm{f^n}$ for l in H. The action label for a field read is written $l.\mathrm{f^n} \rightharpoonup H(l)(\mathrm{f^n})$. Normal field writes $l.\mathrm{f^n} := v$ update $H(l)(\mathrm{f^n})$ to point to v, written $H[(l, \mathrm{f^n}) \mapsto v]$. The action label for a field write is $l.\mathrm{f^n} \leftharpoonup v$. Reads from or writes to **null** raise an exception. Exclusive field reads and writes are similar, except that the permissions held by the current task are updated to indicate a borrow and to remove any pending borrows, respectively, as described in the typing rules above. Remits also remove a pending borrow as described above, and have an empty action label, while **new** consumes exclusive permissions to the locations assigned to the exclusive fields of the new object, as discussed above, and is labeled with $\mathbf{new}(l; \vec{\mathrm{f^X}} \mapsto \vec{v}, \vec{\mathrm{f^n}} \mapsto \vec{v'})$, where the v_i and v'_i are the values assigned to $\mathrm{f}^{\mathrm{X}}_i$ and $\mathrm{f}^{\mathrm{n}}_i$, respectively.

Each form of acquire has two rules, one for failure and one for success. An exclusive acquire on l succeeds if $H(l)(\mathbf{p}) \neq l : \mathrm{X}$, meaning that l has an exclusive permission available for dynamic acquisition. Similarly, a non-exclusive acquire of $\vec{Y}$ on l succeeds if permission set $\Pi, H(l)(\mathbf{p})$ can be separated into $\Pi, l : w\vec{Y}, \Pi_v$, which combines the permissions Π held by the current task, the requested permissions, and some leftover permissions Π_v remaining in $H(l)(\mathbf{p})$. The current permission set Π is included here to allow read-write permissions to be acquired when the current task holds all the private read permissions to an object, while the side condition on Π_v ensures that not all of the remaining permissions are acquired. If these conditions cannot be satisfied, then either acquire throws an exception. Successful acquires of $\vec{\phi}$ for l are labeled with $\mathbf{acq}(l : \vec{\phi})$. Note that acquires on **null** automatically succeed, since $H(\mathbf{null})(\mathbf{p})$ equals any permission set that satisfies the above.

A release of $l : \vec{\phi}$ gives these permissions back to $H(l)(\mathbf{p})$, and is labeled with $\mathbf{rel}(l : \vec{\phi})$. A release on **null** effectively does nothing. An **async** creates a new task with some id t' inside the most closely containing **finish**, and is labeled with $\mathbf{async}(t')$. Rules E-Async and E-AsyncPar handle an **async** in the main body or in a parallel task, respectively, of a **finish**. Parallelism is modeled by allowing steps in the parallel tasks of a **finish**, as captured by E-Par. Finally, a **finish** completes when the main body and all parallel tasks are results, and all permissions returned by the parallel tasks are collected in the parent task. This is labeled with $\mathbf{finish}(\vec{t})$ where $\vec{t}$ are the ids of the completed tasks.

At the top level, small-step evaluation is applied to *machine states* $H \setminus T$, where T is called the top-level task. If $T = (\Pi \setminus res)^t$, then $H \setminus T$ is called a *final state*. We write *Mach* for machine states, and define the set $\mathbf{perms}_l(Mach)$ of permissions for l held by *Mach* as the combination (using ",") of

$$\{\, \Pi|_l \mid \exists(M,t).(\Pi \setminus M)^t \sqsubseteq \mathit{Mach} \,\} \cup H(l)(\mathbf{p}) \cup$$
$$\{\, l : w\Upsilon \mid l' : w\Upsilon - \vec{B} \in \mathbf{perms}_{l'}(\mathit{Mach}) \wedge H(l')(\mathrm{f^X}) = l \wedge \nexists(i,z).B_i = \mathrm{f^X}@z \,\}$$

where $\sqsubseteq$ is the subterm relation. This set is well-defined iff reachability under $\mathrm{f^X}$ fields is acyclic, which is ensured by machine well-formedness for $\mathit{Mach} = H \setminus T$:

1. $\vdash T : \Pi_\mathrm{o}$ for some Π_o and $\cdot \vdash H(l^C)(f) : \Sigma(C)(f)$ for all $l \in \mathrm{Dom}(H)$;
2. $l : \mathrm{X} \geq \mathbf{perms}_l(\mathit{Mach})$;
3. Any **async** occurring in *Mach* is a subterm of a **finish**.

This judgment is written $\vdash \mathit{Mach}$. The first condition ensures that the top-level task and all field values are well-typed. The second ensures that the total permissions in the program to any l are at most X; i.e., there are no duplicated permissions. The final condition ensures that tasks are always spawned inside of a **finish** scope; note that the HJ runtime does this implicitly. Using this definition, we can prove Type Soundness using Preservation and Progress:

Lemma 1 (Preservation). *If* $\vdash \mathit{Mach}$ *and* $\mathit{Mach} \to \mathit{Mach}'$ *then* $\vdash \mathit{Mach}'$.

Lemma 2 (Progress). *If* $\vdash \mathit{Mach}$ *then either Mach is a final state or* $\mathit{Mach} \to \mathit{Mach}'$ *for some* Mach'.

We write $s : \mathit{Mach}_1 \xrightarrow{a} \mathit{Mach}_2$ to denote that s is a step, or derivation of $\mathit{Mach}_1 \xrightarrow{a} \mathit{Mach}_2$. A collection of such steps $\mathit{Mach}_1 \to \ldots \to \mathit{Mach}_n$ is called an *execution*; we write $\mathcal{E} : \mathit{Mach}_1 \dot{\to}^* \mathit{Mach}_n$ to denote that $\mathcal{E}$ is such an execution. We also write $\leq_\mathcal{E}$ for the sequence order of the steps in $\mathcal{E}$. In order to prove that any execution has no data races, we shall prove that any steps which conflict must be ordered by the happens-before order. We define these concepts below. To define this notion, we first define when permissions conflict, which intuitively means that they cannot be held at the same time.

Definition 1 (Conflicting Permissions). *We say that* Π_1 *and* Π_2 conflict, *written* $\Pi \bowtie \Pi'$, *iff* $l : \mathrm{X} \geq \Pi_1|_l, \Pi_2|_l$ *does* not *hold for some* l. *We say that steps* $s_1 : \mathit{Mach}_1 \xrightarrow{(t_1;\Pi_1):a_1} \mathit{Mach}_1'$ *and* $s_2 : \mathit{Mach}_2 \xrightarrow{(t_2;\Pi_2):a_2} \mathit{Mach}_2'$ *conflict, written* $s_1 \bowtie s_2$, *iff* $\Pi_1 \bowtie \Pi_2$.

Definition 2 (Happens-Before). *Step* $s_1 : \mathit{Mach}_1 \xrightarrow{(t_1;\Pi_1):a_1} \mathit{Mach}_1'$ happens-before *step* $s_2 : \mathit{Mach}_2 \xrightarrow{(t_2;\Pi_2):a_2} \mathit{Mach}_2'$, *written* $s_1 \preceq s_2$, *iff* $s_1 \leq_\mathcal{E} s_2$ *and:*

- $t_1 = t_2$;
- $a_1 = \mathbf{async}(t_2)$;
- $a_2 = \mathbf{finish}(\vec{t'}, t_2, \vec{t''})$;
- $a_1 = \mathbf{rel}(l : \vec{\phi})$ *and* $a_2 = \mathbf{acq}(l : \vec{\phi'})$ *for* $l : \vec{\phi} \bowtie l : \vec{\phi'}$; *OR*
- $s_1 \preceq s' \preceq s_2$ *for some* s'.

Theorem 1 (Race-Freedom). *If* $\mathcal{E} : \mathit{Mach} \dot{\to}^* \mathit{Mach}'$ *for* $\vdash \mathit{Mach}$, *and if* $s_1 \bowtie s_2$ *for* $s_1 \leq_\mathcal{E} s_2$, *then* $s_1 \preceq s_2$.

As a final point, we prove that the implementation of acquires and releases does not have to track the permission words that are returned by acquires, and only needs to count the number and sorts of acquires and releases:

Lemma 3. *If* $\vdash \cdot \setminus T$ *and* $\mathcal{E} : \cdot \setminus T \dot{\rightarrow}^{*} H \setminus T'$ *then there is a one-to-one mapping between releases in* $\mathcal{E}$ *and subsequent acquires of the same permissions, where* **new** *is considered an exclusive acquire.*

6 Extensions

We now show how Core HJp can be extended to support two common parallel patterns, array-based parallel loops and objects guarded by critical sections. These extensions modify Core HJp very little, and it is straightforward to show that they preserve the race-freedom property proved in Section 5.

6.1 Array-Based Parallelism

For the purposes of this paper, *array parallelism* is the technique of dividing an array into disjoint pieces which are then modified by parallel tasks. (Parallel tasks that read from the same array are straightforward to support using shared read permissions.) One technical difficulty in supporting array parallelism in HJp is the potential aliasing inherent to standard Java arrays. Specifically, each dimension of a standard Java array is an array of pointers to the next dimension, and there is no guarantee that these pointers do not alias. Thus there is no easy way to break a standard multi-dimensional Java array into disjoint pieces. To address this problem, HJ includes a construct called an *array view* [27,22], which intuitively is an array that is indexed by either one- or many-dimensional *points*. Under the hood, array views are implemented as maps from points to indexes in a one-dimensional Java array. In Core HJp, array views are modeled as maps from points to store locations. Holding a permission ϕ for an array view A is then just a shorthand for holding permission ϕ for all the store locations in A.

To support array parallelism, HJp allows an exclusive permission to an array to be split into exclusive permissions to disjoint pieces of the array, which can then be passed to child tasks for parallel modification. When all the child tasks are done, these exclusive permissions are then combined back into an exclusive permission for the entire array. This is written in Core HJp as follows:

$$\textbf{foreach}\ (x \in r;\ y_1 \subseteq a_1;\ \ldots;\ y_n \subseteq a_n)\ \mathit{body}$$

This expression forks child tasks, as per **async**, to modify disjoint portions of the array views a_1 through a_n, and then waits for the child tasks to complete, as per **finish**. One task is created for each point specified by r, a region expression, and this task executes *body* with the variable x bound to the selected point in r and with each variable y_i is bound to a sub-view of a_i. These sub-views are formed by logically dividing the regions of each array view a_i into rectangular pieces, one for each point in r, and then binding y_i to the sub-view of a_i for the rectangular piece given by the current value of x. For example, the code

$$\textbf{foreach}\ (x \in [1:M, 1:N];\ \mathit{sub} \subseteq a)\ \textbf{for}\ (p \in \mathit{sub}.\mathsf{rgn})\ \mathit{sub}[p] := F(\mathit{sub}[p])$$

modifies 2-dimensional array view a with $M * N$ parallel tasks, each of which applies function F to the elements of a portion of a. The expression $sub.\mathsf{rgn}$ returns the region of the array view sub, while $[1 : M, 1 : N]$ is the rectangular region of points whose dimensions are from 1 to M and from 1 to N, inclusive.

When a **foreach** begins, the parent task must hold exclusive permissions to each array view a_i. Each child task then receives exclusive permissions to the sub-views passed to it in the variables y_i, permissions that it must still hold upon completion. Once all children have completed, the **foreach** then returns the original exclusive permissions for the array views a_i to the parent task.

Permissions can also be stored in and borrowed from the cells of array views just as with normal objects. The one caveat is that only one permission may be borrowed from a given array view at a time. Allowing multiple borrows from the same array view would require complex typing features, such as dependent types, to prove statically that these borrows use different points. This restriction has not been a problem in practice.

6.2 Objects Guarded by Critical Sections

Another common parallel pattern that is supported by HJp is objects that can only be accessed inside critical sections. Critical sections in HJ and HJp are written with the construct **isolated**$(\vec{x})$ M, which is called an isolated region. This indicates that program term M should be run in a way that is isolated from, meaning not at the same time as, any other conflicting isolated region. Two isolated regions are said to conflict if, at runtime, the values of their variables $\vec{x}$ overlap. Isolated regions are therefore similar to locks and to Java **synchronized** statements. A key difference is that the **isolated** construct prevents deadlock: if an isolated region occurs inside another isolated region, then the variables $\vec{x}$ of the inner isolated region must refer, at runtime, to a subset of a the objects referred to by the outer isolated region. Otherwise, an exception is thrown. This is similar to requiring that a task cannot grow the set of locks it holds while already holding locks. Note that throwing an exception in a potential deadlock situation is a conscious choice in the design of HJp. Although there are type systems to statically prevent deadlocks without exceptions [11], we have found that our approach is intuitive to use and does not cause problems in practice.

Isolated regions act as guards in HJp, allowing unrestricted access to the objects referred to by the variables $\vec{x}$, while preventing any access to these objects outside isolated regions that refer to them. To support this in HJp, any class C can be designated as an *isolated class*, meaning that all objects o of class C can only be accessed inside critical sections for o. Isolated classes are designated in Surface HJp by making them subclasses of the `IsolatedObject` class. The **isolated**$(\vec{x})$ M construct then requires the variables $\vec{x}$ to each have type C for some isolated class C. The body M is executed with write permissions $w_i\mathrm{R}, w_i\mathrm{W}$ for each x_i, for some permission words $\vec{w}$.

This can be modeled in Core HJp with acquires and releases of write permissions to the variables $\vec{x}$ at the beginnings and ends of isolated regions. The only change required to Core HJp is that, for isolated objects, acquires never throw

exceptions, they simply wait until the acquire can succeed. This change obviously does not violate the race-freedom guarantee from Section 5, since it only restricts the possible executions that must be considered. As per the discussion in Section 4, however, the HJp compiler does *not* insert acquires and releases for isolated objects, since this would change the synchronization behavior of a program. Instead, if accesses are made to an isolated object outside of an isolated region, the HJp compiler flags a compile-time error.

7 Practical Experience using HJp

Surface HJp is an extension of HJ [14], which itself is an extension of Java to include the **async**, **finish**, and **isolated** constructs, along with a number of constructs for manipulating array views [27,22]. Surface HJp adds the keywords **reading**, **writing**, **shared_reading**, and **exclusive**, which can be applied to a method argument indicate that the corresponding permission to the argument must be held on entry to and exit from the method. The same keywords can also be applied to an entire method, indicating that the permission must be held for **this**. A method can be annotated with `exclusive_ret` to indicate that exclusive permissions are held for the return value on exit. Other keywords are possible, but these are the common cases that were needed for our benchmarks. Object fields and array view element types can be annotated with **exclusive** to indicate storable permissions. To indicate that a class is an isolated class, it must inherit from `IsolatedObject`. Acquires are written with the methods `acquireR()`, `acquireW()`, `acquireSR()`, or `acquireX()`; similar methods exist for releases.

By design, Surface HJp allows the programmer to think only about read, write, shared read, and exclusive permissions, without having to worry about the complexities of permission words, word variables ξ, etc. Specifically, exclusive field reads are not marked with the permissions that are being borrowed, there are no remits, and acquires do not involve let-bindings for permission words. To support this, the HJp compiler infers all of these; as discussed in Section 4, it also inserts acquires and releases, to support gradual typing. We implement this inference using standard dataflow analysis techniques, using a backwards dataflow to determine where permissions must be acquired or borrowed and a forwards dataflow to insert remits and releases. Although we leave the question of completeness of this inference algorithm for future work, it has worked well in practice. This is all implemented as a lightweight compiler pass on the Soot intermediate language [31] used in the back-end of the existing HJ compiler.

In previous work [34], we examined the performance impact of dynamic permission acquires and releases, which yielded an average slowdown of 1.5×. (This work used a subset of the benchmarks we use here.) In essence, that work represented the minimum possible programmer effort in using HJp. Here, we quantify the maximum possible programmer effort, where programs have been modified enough to remove all acquires and releases; this is checked with the `-staticperms` HJp compiler flag. More specifically, we have taken a set of HJ programs, written without permissions in mind, and ported them to HJp by adding enough annotations to statically guarantee race-freedom. With a few exceptions described

Table 1. Programmer Effort for Statically Verifying Race-Freedom in HJp

Benchmark Name	Code Size (Methods)	LoC for array views		LoC for method keywords		LoC for storable perms		LoC for isolated objects		Total	
NPB.CG	1070 (61)	4	0.37%	25	2.33%	7	0.07%	0	0.00%	36	3.36%
JGF.Series	225 (15)	3	1.33%	6	2.67%	3	1.33%	0	0.00%	12	5.33%
JGF.LUFact	467 (20)	0	0.00%	16	3.40%	11	2.36%	0	0.00%	27	5.78%
JGF.SOR	175 (12)	1	0.57%	6	3.43%	4	2.28%	0	0.00%	11	6.29%
JGF.Moldyn	741 (57)	19	2.56%	9	1.20%	29	3.91%	0	0.00%	57	7.69%
JGF.RayTracer	810 (67)	1	0.12%	57	6.75%	22	2.60%	4	0.47%	84	9.94%
BOTS.NQueens	95 (3)	0	0.00%	3	3.15%	0	0.00%	1	1.00%	3	3.16%
BOTS.Fibonacci	70 (3)	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
BOTS.FFT	4480 (46)	13	0.29%	33	0.74%	0	0.00%	0	0.00%	46	1.04%
PDFS	537 (26)	0	0.00%	10	1.80%	8	1.44%	8	1.44%	26	4.67%
DPJ.BarnesHut	682 (56)	1	0.15%	18	2.64%	10	1.47%	0	0.00%	28	4.11%
DPJ.MonteCarlo	2877 (287)	3	0.10%	151	5.25%	22	0.59%	1	0.03%	177	6.15%
DPJ.IDEA	228 (18)	1	0.43%	9	3.94%	8	3.50%	0	0.00%	18	7.89%
DPJ.CollisonTree	1032 (69)	0	0.00%	108	10.40%	24	2.32%	0	0.00%	132	12.70%
DPJ.K-Means	501 (38)	1	0.20%	25	4.99%	6	1.20%	1	0.20%	33*	6.59%
Total	13990 (778)	47	0.33%	476	3.40%	152	1.09%	15	0.11%	690	4.93%

below, the resulting HJp programs compile to the exact same programs as the original HJ programs, so there is guaranteed to be no performance penalty.

We chose a number of small- to large-scale parallel benchmarks from the NAS Parallel Benchmark suite [3] (NPB), JavaGrande benchmark suite [30] (JGF), the BOTS benchmark suite [17], a Parallel Depth First Search application (PDFS), and the benchmarks used for Deterministic Parallel Java [6,7] (DPJ), another type system for statically ensuring race-freedom. The DPJ benchmarks were originally written in Java and were ported to HJ, while the others were originally written in HJ. We measured the number of lines of code (LoC) that had to be modified (from the HJ version) to statically ensure race-freedom.

The results of this experiment are summarized in Table 1, which presents, for each benchmark, the name prefixed with the suite it came from, the code size in LoC with the number of methods in parentheses, and then the LoC modified from the original code. The latter are divided into modifications needed for: array parallelism, which mostly included adding the **foreach** construct from Section 6.1; adding the method keywords discussed above; adding the **exclusive** keyword to fields and array view elements types, to use storable permissions; and designating classes as subclasses of `IsolatedObject`. On average, HJp requires about 5% of the LoC to be annotated, with the majority of the annotations, accounting for 3.4% of the LoC, being method keywords.

There is one additional modification that was necessary for the K-Means DPJ benchmark, which is not reflected in Table 1; the total LoC for this benchmark is marked with an asterisk. The issue is that the original code uses an array `lock[]`

Table 2. Comparison of Programmer Effort and Runtimes between DPJ and HJp

Benchmark Name	Code Size (Methods)	LoC modified in DPJ		LoC modified in HJp		Execution Time (s) Xeon		T2	
						DPJ	HJp	DPJ	HJp
DPJ.BarnesHut	682 (56)	80	11.73%	28	4.11%	4.207	4.041	5.715	5.695
DPJ.MonteCarlo	2877 (287)	220*	7.64%	177	6.15%	3.102	3.047	6.065	5.792
DPJ.IDEA	228 (18)	24	10.52%	18	7.89%	0.725	0.731	0.737	0.705
DPJ.CollisonTree	1032 (69)	233	22.58%	132	12.70%	1.253	1.268	3.282	3.245
DPJ.K-Means	501 (38)	5*	1.00%	33*	6.59%	20.188	19.016	65.084	64.953
Total	5320 (468)	557	10.47%	388	7.29%				

of locks, where each `lock[i]` guards accesses to the elements `new_centers[i]` and `globalSize[i]` of two other arrays. To support this access pattern in HJp, we had to refactor these into a single array of a new class `ClusterAttr` that inherits from `IsolatedObject`. This class has two fields, `new_centers` and `globalSize`, where the first is marked as an **exclusive** field to allow it to be accessed during an isolated region **isolated(x)** for the parent `ClusterAttr` object `x`. The second field, `globalSize`, is a primitive Java integer, and so does not need to be **exclusive**.

We also directly compared HJp with DPJ on the DPJ benchmarks. This comparison is summarized in Table 2. On average, HJp required modification to only 7.3% of the LoC, while DPJ required modification to 10.5% of the LoC. For most benchmarks, HJp requires fewer annotations; for K-Means, however, HJp requires annotations on 33 LoC, as opposed to the 5 LoC for DPJ. Both the K-Means and MonteCarlo benchmarks, however, are not completely verified by DPJ: each of these benchmarks require the user to add a `commutative` annotation to one of the methods. This is an unchecked user assertion in DPJ stating that two parallel executions of the method always commute. Thus, although the HJp version of K-Means requires more annotations, this version is statically verified. Further, the HJp version of the MonteCarlo benchmark is also statically verified, and requires fewer annotations than DPJ.

The execution times of DPJ and HJp were also compared, to ensure that there is nothing about HJp that limits performance. The results are given on the right side of Table 2. This includes numbers for two machines: a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory, running Red Hat Linux (RHEL 5) and Sun JDK 1.6 (64-bit version); and a 128-thread (dual-socket, 8 cores per socket, 8 threads per core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory, running Solaris 10 and Sun JDK 1.6 (64-bit version). The size of the thread pool for DPJ was varied, and the table shows the best numbers obtained. In most cases, DPJ and HJp performed comparably. For K-Means, HJp performed better; we believe this is because the DPJ runtime uses JUC locks, while HJp uses the built-in **synchronized** construct of Java, which is significantly faster.

8 Related Work

There has been much recent work on imperative parallel programming languages that prevent data races, mostly based either on *ownership* or *permissions*. In the former, each object has an *owner*, specified in its type, that mediates all accesses to the object. Originally introduced by Clarke et al. [15] to control aliasing, Boyapati et al. [11] showed how to use ownership to ensure race-freedom by only allowing accesses to an object when the current task either owns an object or holds a lock that owns an object. Static race detection [25,1,19,2] is form of ownership, where each object is either owned by a lock or by the current task. In the work of Vaziri et al. [33,32], object fields are owned by atomic set objects, which ensure that all sequences of accesses to a group of related values satisfy a strong consistency guarantee called *atomic-set serializability*. Deterministic Parallel Java (DPJ) [7,6] also fits the ownership model, where each object is owned by a memory region. DPJ ensures determinism by restricting parallelism to disjoint regions.

Though it is a powerful notion, ownership suffers from a number of drawbacks. First, it requires programmer annotations to specify ownership; e.g., the comparison of HJp with DPJ in Section 7 indicates a higher annotation burden in DPJ. Second, ownership-based systems are only designed for a single parallel pattern; e.g., traversal-based ownership in DPJ and lock-based ownership in other approaches. The work of Vaziri et al. [33,32] partially addresses this concern, since all synchronizations are automatically generated by the compiler after the atomic sets are specified. A final issue is the static nature of ownership. This means that the synchronization behavior of an object cannot change over time, which again limits the algorithms that can be written using ownership.

Permission-based systems, in contrast, view the ability to access an object as a resource, which may change over time. They are closely related to linear type systems, which ensure that resources are not duplicated or deleted when doing so is disallowed. A number of systems for avoiding races have been based on linear types, since only one task can have permission on a linear pointer at a time. Haller and Odersky [20] describe one such system, Scala capabilities. A major breakthrough was Boyland's work on fractional permissions [12], which showed how a linear read/write permission could be split into fractional read permissions. One approach that builds on this work is typestate-oriented programming (TSOP) [35,4,5], in which the "state" of an object may be changed only when an exclusive, non-fractional permission is held for it. Beckman et al. [4] use this approach to ensure that state changes do not cause data races. Although gradual typing has been studied for TSOP [35], it is not clear that this could be directly applied to race-freedom as in HJp. In addition, permissions have been studied in the context of program verification using separation logic [16,10,9].

The storable permissions of HJp can be seen as a restricted form of Boyland's nested permissions [13]. Although storable permissions are less expressive, they do seem to correspond to many of Boyland's examples in a more concise way. Specifically, storable permissions allow these examples to be expressed without

using existentials, object equality, and disjunction at the type level, which seem to be required to express them using nested permissions. Fahndrich and DeLine [18] have also introduced a notion of permission guards, where permission "key" ρ allows access to a linear permission τ. The latter can be temporarily borrowed using the "focus" construct when permission key ρ is held, in a manner similar to permission borrowing for exclusive fields in HJp.

9 Conclusions

In this paper, we present a new type system for race-free parallel programming, based on Boyland's fractional permissions. Our system, Habanero Java with permissions (HJp), is an extension of the Habanero Java (HJ) task-parallel language. HJp is designed to be *gradual*, meaning that it can compile parts of a program that do not contain any annotations or types related to race-freedom, by inserting dynamic checks. This allows existing programs to be compiled with no modifications. The programmer can then gradually add permission annotations to increase performance and static guarantees, eventually leading to a fully annotated, race-free program. Further, no parallel or concurrent programming expertise is necessary to understand these permission annotations. We demonstrate how a number of different concurrency patterns, such as fork-join, array partitioning, and objects guarded by critical sections, can be accommodated in HJp. We also introduce a number of theoretical advances over previous work on fractional permissions, including aliased write permissions and simpler way to store permissions in objects than previous approaches. Finally, we evaluate the annotation burden required to yield statically-verified race-free benchmarks in HJp starting from existing HJ benchmarks, using a complete implementation of the compiler and runtime of the HJp type system. Our results show that for 15 benchmarks we have been able to statically verify race-freedom with only a modest number (5% of the lines of code on average) of annotations.

Acknowledgments. We would like to acknowledge the generous help of John Boyland for his many comments and suggestions.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for java. ACM Trans. Program. Lang. Syst. 28, 207–255 (2006)
2. Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of java without data races. In: OOPSLA (2000)
3. Bailey, D.H., et al.: The NAS parallel benchmarks. Intl. Journal of Supercomputer Applications 5(3) (1994)
4. Beckman, N.E., Bierhoff, K., Aldrich, J.: Verifying correct usage of atomic blocks and typestate. In: OOPSLA 2008 (2008)
5. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA 2007 (2007)

6. Bocchino, R.L., et al.: A type and effect system for deterministic parallel java. In: OOPSLA 2009 (2009)
7. Bocchino, R.L., et al.: Safe nondeterminism in a deterministic-by-default parallel language. In: POPL 2011 (2011)
8. Boehm, H.-J., Adve, S.V.: Foundations of the c++ concurrency memory model. In: PLDI (2008)
9. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL (2005)
10. Bornat, R., Calcagno, C., Yang, H.: Variables as resource in separation logic. Electron. Notes Theor. Comput. Sci. 155, 247–276 (2006)
11. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA 2002 (2002)
12. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
13. Boyland, J.: Semantics of fractional permissions with nesting. ACM Trans. Program. Lang. Syst. 32 (2010)
14. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-java: the new adventures of old X10. In: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java, PPPJ (2011)
15. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA (1998)
16. Dodds, M., Jagannathan, S., Parkinson, M.J.: Modular reasoning for deterministic parallelism. In: POPL 2011 (2011)
17. Duran, A., et al.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: ICPP (2009)
18. Fahndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: PLDI (2002)
19. Flanagan, C., Freund, S.N.: Type-based race detection for java. In: PLDI (2000)
20. Haller, P., Odersky, M.: Capabilities for Uniqueness and Borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010)
21. Heule, S., et al.: Fractional permissions without the fractions. In: Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs, FTfJP (2011)
22. Joyner, M.: Array Optimizations for High Productivity Programming Languages. PhD thesis, Rice University (2008)
23. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: POPL 2005 (2005)
24. Marino, D., et al.: DRFx: a simple and efficient memory model for concurrent programming languages. In: Proceedings of PLDI 2010 (2010)
25. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: PLDI (2006)
26. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. 18, 865–911 (2008)
27. Shirako, J., Kasahara, H., Sarkar, V.: Language Extensions in Support of Compiler Parallelization. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 78–94. Springer, Heidelberg (2008)
28. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (2006)
29. Singh, A., et al.: Efficient processor support for DRFx, a memory model with exceptions. In: Proceedings of ASPLOS 2011 (2011)
30. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (2001)

31. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a java optimization framework. In: Proceedings of CASCON 1999 (1999)
32. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL (2006)
33. Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J.: A Type System for Data-Centric Synchronization. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 304–328. Springer, Heidelberg (2010)
34. Westbrook, E., Zhao, J., Budimlić, Z., Sarkar, V.: Permission regions for race-free parallelism. In: RV (2011)
35. Wolff, R., Garcia, R., Tanter, É., Aldrich, J.: Gradual Typestate. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 459–483. Springer, Heidelberg (2011)

Verification of Snapshot Isolation in Transactional Memory Java Programs

Ricardo J. Dias[1], Dino Distefano[2], João Costa Seco[1], and João M. Lourenço[1,⋆]

[1] CITI, Universidade Nova de Lisboa, Portugal
{rjfd,joao.seco,joao.lourenco}@di.fct.unl.pt
[2] Queen Mary University of London, UK
ddino@eecs.qmul.ac.uk

Abstract. This paper presents an automatic verification technique for transactional memory Java programs executing under snapshot isolation level. We certify which transactions in a program are safe to execute under snapshot isolation without triggering the *write-skew* anomaly, opening the way to run-time optimizations that may lead to considerable performance enhancements.

Our work builds on a novel deep-heap analysis technique based on separation logic to statically approximate the read- and write-sets of a transactional memory Java program.

We implement our technique and apply our tool to a set of micro benchmarks and also to one benchmark of the STAMP package. We corroborate known results, certifying some of the examples for safe execution under snapshot isolation by proving the absence of *write-skew* anomalies. In other cases our analysis has identified transactions that potentially trigger previously unknown *write-skew* anomalies.

1 Introduction

Full-fledged Software Transactional Memory (STM) [18,11] usually provides strict isolation between transactions and full serializability semantics. Alternative relaxed semantics approaches, based on weaker isolation levels that allow transactions to interfere and to generate non-serializable execution schedules, are known to perform considerably better in some cases. The interference among non-serializable transactions are commonly known as *serializability anomalies* [2].

Snapshot Isolation (SI) [2] is a well known relaxed isolation level widely used in databases, where each transaction executes with relation to a private copy of the system state — a snapshot — taken at the beginning of the transaction and stored in a local buffer. All write operations are kept pending in the local buffer until they are committed in the global state. Reading modified items always refer

⋆ This work was partially supported by the Euro-TM EU COST Action IC1001, the Portuguese national research projects RepComp (PTDC/EIA-EIA/108963/2008), Synergy-VM (PTDC/EIA-EIA/113613/2009) and StreamLine (PTDC/EIA-CCO/104583/2008), and the research grant SFRH/BD/41765/2007. Distefano was supported by the Royal Academy of Engineering.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 640–664, 2012.

to the pending values in the local buffer. In all cases, committing transactions obey the general First-Commiter-Wins rule. This rule states that a transaction A can only commit if no other concurrent transaction B has committed modifications to data items pending to be committed by transaction A. Hence, for any two concurrent transactions modifying the same data item, only the first one to commit will succeed.

Tracking memory operations introduces some overhead, and TM systems running under serializable isolation level must track both memory read and write accesses, incurring in considerable performance penalties. Validating transactions in SI only requires to check if any two concurrent transaction wrote at a common data item. Hence the runtime system only needs to track the memory write accesses per transaction, ignoring the read accesses, possibly boosting the overall performance of the transactional runtime, as shown in [7].

Although appealing for performance reasons, the application of SI may lead to non-serializable executions, resulting in a serializability anomaly called *write-skew*. For instance, a *write-skew*, arises in the following example of two statements running in concurrent transactions

$$x := x + y \quad || \quad y := y + x$$

In this case, it is possible to find a trace of execution that is not serializable and yields unexpected results. In general, this anomaly occurs when two transactions are writing on disjoint memory locations (x and y) but are also reading data that is being modified by the other.

In this paper we present a verification technique for STM Java programs that statically detects if any two transactions may cause a *write-skew* anomaly. The application of the proposed technique may be used to optimize program execution, by letting memory transactions run in snapshot isolation whenever possible, and by explicitly requiring the full serializability semantics otherwise. Our technique performs deep-heap analysis (also called shape analysis) based on separation logic [16,8] to compute memory locations in the read- and write-sets for each distinguished transaction in a Java program. The analysis only requires the specification of the state of the heap for each transaction and is able to automatically compute *loop invariants* during the analysis. Our analysis computes read and write-sets of transactions using *heap paths*, which capture dereferences through field labels, choice and repetition.

For instance, a *heap path* of the form $x.(\mathit{left} \mid \mathit{right})^*.\mathit{right}$ describes the access to a field labeled *right*, on a memory location reachable from variable x after a number of dereferences through *left* or *right* fields.

We implemented a tool with the proposed techniques, called StarTM, which allows to analyze Java Bytecode programs extended with STM annotations. To validate our approach, we tested implementations of a transactional Linked List and of a transactional Binary Search Tree, and also of a Java implementation of the STAMP Intruder benchmark [5]. Our results confirm that i) it is possible to safely execute concurrent transactions of a Linked List under snapshot isolation with noticeable performance improvements, supporting the arguments of

[17]; ii) it is possible to build a transactional insert method in a Binary Search Tree that is safe to execute under SI; and iii) our automatic analysis of the STAMP Intruder benchmark found a new *write-skew* anomaly in the existing implementation.

We impose some limitations on the programs for which our approach is able to guarantee the absence of *write-skew* anomalies. We only support acyclic data structures, such as tree-like data structures, and only detect *write-skews* between pairs of transactions.

The main contributions of this paper are:

- The first program verification technique to statically detect the *write-skew* anomaly in transactional memory programs;
- The first technique able to verify transactional memory programs even in presence of deep-heap manipulation thanks to the use of shape analysis techniques;
- A model that captures fine-grained manipulation of memory locations based on *heap paths*;
- An implementation of our technique and the application of the tool to a set of intricate examples.

The remainder of the paper describes the theory of our analysis technique and the validation experiments. We start by describing a step-by-step example of applying StarTM to a simple example in section 2. We then present the core language, in section 3, and the abstract domain for the analysis procedure in section 4. In section 5, we present the symbolic execution of programs against the abstract state representation. We finalize the paper by presenting some experimental results in Section 6 and comparing our approach with others in Section 7.

2 StarTM by Example

StarTM analyzes Java multithreaded programs that make use of memory transactions. The scope of a memory transaction is defined by the scope of a Java method annotated with **@Atomic**, which in our case requires a mandatory argument with an abstract description of the initial state of the heap. Other methods called inside a transactional method do not require this initial description, as it is automatically computed by the symbolic execution.

To describe the abstract state of the heap, we use a subset of separation logic formulae composed of a set of predicates — among which a *points-to* ($\mapsto$) predicate — separated by the special separation conjunction ($*$) typical of separation logic. The user can define new predicates in a proper scripting language and also define abstraction functions which, in case of infinite state spaces, allows the analysis to converge. The abstraction function is defined by a set of abstraction rules as in the jStar tool [9]. The user defined predicates and abstraction rules are described in separate files and are associated with the transactions' code by the class annotations **@Predicates** and **@Abstractions**, which receive as argument the corresponding file names.

```
1  @Predicates(file="list_pred.sl")
2  @Abstractions(file="list_abs.sl")
3  public class List { public class Node{ ... } ... }
```

```
23 @Atomic(state= "| this -> [head:h']
24         * List(h', nil)")
25 public void add(int value) {
26     boolean result;
27     Node prev = head;
28     Node next = prev.getNext();
29     while (next.getValue() < value) {
30         prev = next;
31         next = prev.getNext();
32     }
33     if (next.getValue() != value) {
34         Node n = new Node(value, next);
35         prev.setNext(n);
36     }
37 }
```

```
39 @Atomic(state= "| this -> [head:h']
40         * List(h', nil)")
41 public void remove(int value) {
42     boolean result;
43     Node prev = head;
44     Node next = prev.getNext();
45     while (next.getValue() < value) {
46         prev = next;
47         next = prev.getNext();
48     }
49     if (next.getValue() == value) {
50         prev.setNext(next.getNext());
51     }
52 }
```

Fig. 1. Order Linked List code

```
// list_pred.sl file

/*** Predicate definition ***/
Node(+x,-n) <=> x -> [next:n] ;;

List(+x,-y) <=> x != y /\
  ( Node(x,y) \/ E z'. Node(x,z') *
     List(z',y) );;
```

```
// list_abs.sl file
/*** Abstractions definition ***/
Node(x, y') * Node(y',z)~~>List(x, z):
    y' nin context;
    y' nin x;
    y' nin z
;;
...
List(x,y') * Node(y',z)~~>List(x, z):
    y' nin context;
    y' nin x;
    y' nin z
;;
```

Fig. 2. Predicates and Abstraction rules of Linked List

We use as running example the implementation of an ordered singly linked list, adapted from the DeuceSTM [13] samples, shown in Fig. 1. The corresponding predicates and abstractions rules are defined in Fig. 2. The predicate `Node(+x,-y)` defined in Fig. 2 by

$$Node(x, y) \Leftrightarrow x \mapsto [next : y]$$

is valid if variable x points to a memory location where the corresponding *next* field points to the same location as variable y, or both the *next* field and y point to nil. Predicate `List(+x,-y)` defined by

$$List(x, y) \Leftrightarrow x \neq y \wedge (Node(x, y) \vee \exists z'.\, Node(x, z') * List(z', y))$$

is valid if variables x and y point to distinct memory locations and there is a chain of nodes leading from the memory location pointed by x to the memory location pointed by y. The predicate is also valid when both y and the last node in the chain point to nil.

```
# Method boolean add(int value)
Result 1:
ReadSet:   { this.head.(next)[*A].next.value }
WriteSet>: { }
WriteSet<: { }

Result 2:
ReadSet:   { this.head.(next)[*B].next.value }
WriteSet>: { this.head.(next)[*B].next }
WriteSet<: { this.head.(next)[*B].next }


# Method boolean remove(int value)
Result 1:
ReadSet:   { this.head.(next)[*C].next.value }
WriteSet>: { }
WriteSet<: { }

Result 2:
ReadSet:   { this.head.(next)[*D].next.value, this.head.(next)[*D].next.next}
WriteSet>: { this.head.(next)[*D].next }
WriteSet<: { this.head.(next)[*D].next }
```

Fig. 3. Sample of StarTM result output for the Linked List example

The modifiers `+` and `-` of the predicate parameters indicate that the corresponding parameter points to a memory location respectively inside or outside of the memory region defined by the predicate. A more precise definition of these modifiers is presented in Section 4.2.

In Fig. 1, we annotate the add(**int**) and remove(**int**) methods as transactions with the initial state described by the following formula:

```
| this->[head:h'] * List(h',nil)
```

This formula states that variable **this** points to a memory location that contains an object of class List, and whose field *head* points to the same memory location pointed by the existential variable[1] h', which is the entry point of a list with at least one element.

StarTM performs an inter-procedural symbolic execution of the program. The abstract domain used by the symbolic execution is composed by a separation logic formula describing the abstract heap structure, and the abstract read- and write-sets. The abstract write-set is defined by two sets: a *may* write-set and a *must* write-set. As the naming implies one over-approximates, and the other under-approximates the possible real write-set. The abstract read-set is an over-approximation of the possible real read-set. The read- and write-sets are defined as sets of *heap paths*. A memory location is represented by its path, in terms of field accesses, beginning from some shared variable. We assume that the parameters of a transactional method and the instance variable `this` are shared in the context of that transaction.

[1] Throughout this paper we consider primed variables as implicitly existentially quantified.

The sample of the results of our analysis, depicted in Fig. 3, includes two possible pairs of read- and write-sets for method add(**int**). The *may* write-set is denoted by label `WriteSet>` and the *must* write-set is denoted by label `WriteSet<`. The first result has an empty write-set[2], and thus corresponds to a read-only execution of the method add(**int**), where the *heap path* in the read-set can be interpreted as follows. The *heap path* `this.head.(next)[*A].next.value` asserts that method add(**int**) reads the *head* field from the memory location pointed by variable *this* and following the memory location pointed by *head* it reads the *next* field, then for each memory location it reads the *next* and *value* fields and hops to the next memory location through the *next* field. In the last memory location accessed it only reads the *value* field. In general, we can interpret the meaning of an abstract read-set as all the memory locations represented by the *heap paths* present in the read-set and also by their prefixes.

The star ($*$) operator has always a label attached, in case of `[*A]`, the label is `A`. This label is used to identify the subpath guarded by the star and can be interpreted, in this case, as $A = (next)^*$. This label is existentially quantified in a pair of read- and write-sets.

The second pair of read- and write-sets of method add(**int**) in Fig. 3 contains the same read-set and a different write-set. In this case the *may* and *must* write-sets are equal. The *heap path* `this.head.(next)[*B].next` asserts that the *next* field, of the memory location represented by the path $this.head.(next)^*_B$, was written.

It is important to notice that the interpretations of the read- and write-set are different. In the read-set we consider that all the path prefixes of all *heap path* expressions were read, while in the write-set we consider that there was a single write operation in the last field of each *heap path* expression.

The *may* write-set may contain *heap paths* of the form $this.head.(next)^{\bar{*}}_B$. In this case, the interpretation of this expression is that the field *next* is written in every memory location represented by the path $this.head.(next)^*_B$. More details on *heap path* expressions are given in Section 4.2.

The analysis also originates two possible results for method remove(**int**). The first result for this method is similar to the first result for method add(**int**). In the second result for method remove(**int**), the field *next* is read for all memory locations including the last memory location where field *value* was accessed, since the star label is the same in the two *heap path* expressions in the read-set. The write-set is the same as in the add(**int**) method.

We can now check for the possible occurrence of a *write-skew* anomaly. We define a *write-skew* condition as:

Definition 1 (Abstract Write-Skew). *Let* T_1 *and* T_2 *be two transactions, and let* $\mathcal{R}_i$, $\mathcal{W}_i^{>}$ *and* $\mathcal{W}_i^{<}$ $(i = 1, 2)$ *be their corresponding abstract read-,* may *write- and* must *write-sets. There is a write-skew anomaly if*

$$\mathcal{R}_1 \cap \mathcal{W}_2^{>} \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{>} \cap \mathcal{R}_2 \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{<} \cap \mathcal{W}_2^{<} = \emptyset$$

[2] If the context is not ambiguous we will always refer to both the *may* and *must* write-sets.

```
# Method boolean remove(int value)
Result 2:
ReadSet:   { this.head.(next)[*D].next.value, this.head.(next)[*D].next.next }
WriteSet>: { this.head.(next)[*D].next, this.head.(next)[*D].next.next }
WriteSet<: { this.head.(next)[*D].next, this.head.(next)[*D].next.next }
```

Fig. 4. Sample of StarTM result output for corrected remove(**int**) method

We will consider that each result (a pair of a read- and a write-set) corresponds to a single transaction instance. From the above condition we may trivially ignore the results with an empty write-set. Hence, only result pairs with non-empty write-sets need to be checked.

We denote the second result of the add(**int**) method as T_{add}, and the second result of the remove(**int**) method as T_{rem}. To detect the possible existence of a *write-skew* we need to check the following pairs:

$$(\mathrm{T}_{add}, \mathrm{T}_{add}),\ (\mathrm{T}_{rem}, \mathrm{T}_{rem}),\ (\mathrm{T}_{add}, \mathrm{T}_{rem})$$

Let's examine in detail the pair $(\mathrm{T}_{add}, \mathrm{T}_{rem})$. We simplify the description of the read-set of each transaction by ignoring the field `value`, since neither transactions writes to that field and thus we will focus only on interactions with the field `next`. We assume that the shared variable `this` points to the same object in both transactions, otherwise no conflicts would ever arise. The read- and write-set for transactions T_{add}, and T_{rem} (relative to field *next*) are

$$\begin{aligned}
\mathcal{R}_{add} &= \{\mathit{this.head},\ \mathit{this.head.B},\ \mathit{this.head.B.next}\} \\
\mathcal{W}^{>}_{add} &= \mathcal{W}^{<}_{add} = \{\mathit{this.head.B.next}\} \\
\mathcal{R}_{rem} &= \{\mathit{this.head},\ \mathit{this.head.D},\ \mathit{this.head.D.next},\ \mathit{this.head.D.next.next}\} \\
\mathcal{W}^{>}_{rem} &= \mathcal{W}^{<}_{rem} = \{\mathit{this.head.D.next}\}
\end{aligned}$$

Given these read- and write-sets, if an instantiation of B and D exist that satisfies the *write-skew* condition then the concurrent execution of these two transactions could possibly cause a *write-skew* anomaly. In this particular case, the assertion $B = D.next$, which means that the memory locations represented by B are the same as the ones represented by $D.next$, satisfies the *write-skew* condition.

To correct the list implementation from triggering a *write-skew* anomaly one can add the additional write operation next.setNext(**null**) between lines 50 and 51 of the code shown in Fig. 1. This write operation, although unnecessary in terms of the list semantics, is essential to make the list implementation safe under snapshot isolation as we shall see. Given this new implementation, the result of the analysis by StarTM is depicted in Fig. 4. Notice that the write-set has two *heap paths* describing that the transaction writes the *next* field of the penultimate and last memory locations. Now, the new read- and write-set for transactions T_{add}, and T_{rem} (relative to field *next*) are

$$
\begin{array}{lll}
e & ::= & (\textit{expression}) \\
 & x & (\text{variables}) \\
 & \mid \ \mathsf{null} & (\text{null value}) \\
A & ::= & (\textit{assignments}) \\
 & x := e & (\text{local}) \\
 & \mid \ x := y.f & (\text{heap read}) \\
 & \mid \ x := \mathit{fun}(\vec{y}) & (\text{function call}) \\
 & \mid \ x.f := e & (\text{heap write}) \\
 & \mid \ x := \mathsf{new} & (\text{allocation})
\end{array}
\qquad
\begin{array}{lll}
b & ::= & (\textit{boolean exp}) \\
 & e \oplus_b e & (\text{boolean op}) \\
 & \mid \ \mathsf{true} \mid \mathsf{false} & (\text{bool values}) \\
S & ::= & (\textit{statements}) \\
 & S \ ; S & (\text{sequence}) \\
 & \mid \ A & (\text{assignment}) \\
 & \mid \ \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S & (\text{conditional}) \\
 & \mid \ \mathsf{while}\ b\ \mathsf{do}\ S & (\text{loop}) \\
 & \mid \ \mathsf{return}\ e & (\text{return}) \\
 & \mid \ \mathsf{skip} & (\text{Skip})
\end{array}
$$

$$P ::= \mathit{fun}(\vec{x}) = S \ \mid \ P \ (\textit{program})$$

Fig. 5. Core language syntax for programs

$$
\begin{aligned}
\mathcal{R}_{add} &= \{\mathit{this.head},\ \mathit{this.head.B},\ \mathit{this.head.B.next}\} \\
\mathcal{W}^{>}_{add} &= \mathcal{W}^{<}_{add} = \{\mathit{this.head.B.next}\} \\
\mathcal{R}_{rem} &= \{\mathit{this.head},\ \mathit{this.head.D},\ \mathit{this.head.D.next}, \mathit{this.head.D.next.next}\} \\
\mathcal{W}^{>}_{rem} &= \mathcal{W}^{<}_{rem} = \{\mathit{this.head.D.next},\ \mathit{this.head.D.next.next}\}
\end{aligned}
$$

In this case, it is not possible to find an instantiation for B and D, such that the *write-skew* condition is true. Hence, these transactions can execute concurrently under SI without ever triggering the *write-skew* anomaly.

3 Core Language

In this section we define a core language to support our static analysis. We include the subset of Java that captures essential features such as object creation (new), field dereferencing ($x.f$), assignment ($x := e$), and function invocation ($\mathit{fun}(\vec{x})$). The syntax of the language is defined by the grammar in Fig. 5. A program in this language is a set of function definitions. We do not explicitly represent transactions nor an entry point in the syntax, and we assume that all functions are transactions that can be called concurrently.

We assume a countable set of program variables Vars (ranged over by $x, y, \ldots$), a set of shared variables SVars $\subseteq$ Vars, a countable disjoint set of primed variables Vars$'$ (ranged over by $x', y', \ldots$), a countable set of locations Locations, and a finite set of field names Fields. The operational semantics for the language is defined over configurations of the form $\langle s, h, S\rangle$, where $s \in$ Stacks is a stack (a mapping from variables to values), $h \in$ Heaps is a (concrete) heap (a mapping from locations to values through field labels).

$$
\begin{array}{lll}
e & ::= & (\textit{expressions}) \\
 & x, y, \ldots \in \mathsf{Vars} & (\text{program variables}) \\
 & \mid\ x', y', \ldots \in \mathsf{Vars}' & (\text{existential variables}) \\
 & \mid\ \mathsf{nil} & (\text{null value}) \\
\rho & ::= f_1 : e,\ \ldots,\ f_n : e & (\textit{record}) \\
 & & \\
S & ::= e \mapsto [\rho] \ \mid\ p(\vec{e}) & (\textit{spatial predicates}) \\
P & ::= e = e & (\textit{pure predicates}) \\
\Pi & ::= \mathsf{true} \ \mid\ P \wedge \Pi & (\textit{pure part}) \\
\Sigma & ::= \mathsf{emp} \ \mid\ S * \Sigma & (\textit{spatial part}) \\
 & & \\
\mathcal{H} & ::= \Pi | \Sigma & (\textit{symbolic heap})
\end{array}
$$

Fig. 6. Separation logic syntax

$$
\begin{aligned}
\mathsf{Values} &= \mathsf{Locations} \cup \{\mathsf{nil}\} \\
\mathsf{Stacks} &= (\mathsf{Vars} \cup \mathsf{Vars}') \to \mathsf{Values} \\
\mathsf{Heaps} &= \mathsf{Locations} \rightharpoonup_{\mathit{fin}} (\mathsf{Fields} \to \mathsf{Values})
\end{aligned}
$$

The small step structural operational semantics is the standard for this kind of imperative language defined by the reduction relation $\langle s, h, S\rangle \Longrightarrow \langle s', h', S'\rangle$.

4 Symbolic States

In the symbolic execution a *symbolic state* is of the form $(\mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W})$: where $\mathcal{H}$ is a *symbolic heap*, defined using a fragment of separation logic formulae, $\mathcal{M}$ is a map between variables and *heap path* expressions, and $\mathcal{R}$ and $\mathcal{W}$ are read- and write-sets. The write-set $\mathcal{W}$ in our analysis is actually composed by two sets: a *may* write-set, denoted by $\mathcal{W}^{>}$, which over-approximates the concrete write-set, and a *must* write-set, denoted by $\mathcal{W}^{<}$, which under-approximates the concrete write-set.

The fragment of separation logic formulae that we use to describe symbolic heaps is defined by the grammar in Fig. 6. Satisfaction of a formula $\mathcal{H}$ by a stack s and heap h is denoted $s, h \models \mathcal{H}$ and defined by structural induction on $\mathcal{H}$ in Fig. 7. There, $[\![p]\!]$ is as usual a component of the least fixed point of a monotone operator constructed from a inductive definition set; see [3] for details. In this heap model a location maps to a record of values. The formula $e \mapsto [\rho]$ can mention any number of fields in ρ, and the values of the remaining fields are implicitly existentially quantified.

4.1 Symbolic Heaps

Symbolic heaps are abstract models of the heap of the form $\mathcal{H} = \Pi|\Sigma$ where Π is called the *pure part* and Σ is called the *spatial part*. We use prime variables

$$
\begin{array}{rl}
s,h \models \mathsf{emp} & \text{iff } \mathsf{dom}(h) = \emptyset \\
s,h \models x \mapsto [f_1 : e_1, \ , f_n : e_n] & \text{iff } h = [s(x) \mapsto r] \text{ where } r(f_i) = s(e_i) \text{ for } i \in [1,n] \\
s,h \models p(\vec{e}) & \text{iff } (s(\vec{e}), h) \in [\![p]\!] \\
s,h \models \Sigma_0 * \Sigma_1 & \text{iff } \exists h_0, h_1.\ h = h_0 * h_1 \text{ and } s,h_0 \models \Sigma_0 \text{ and } s,h_1 \models \Sigma_1 \\
s,h \models e_1 = e_2 & \text{iff } s(e_1) = s(e_2) \\
s,h \models \Pi_1 \wedge \Pi_2 & \text{iff } s,h \models \Pi_1 \text{ and } s,h \models \Pi_2 \\
\\
s,h \models \Pi | \Sigma & \text{iff } \exists \vec{v'}.\ \left(s(\vec{x'} \mapsto \vec{v'}), h \models \Pi\right) \text{ and } \left(s(\vec{x'} \mapsto \vec{v'}), h \models \Sigma\right) \\
 & \quad \text{where } \vec{x'} \text{ is the collection of existential variables} \\
 & \quad \text{in } \Pi | \Sigma
\end{array}
$$

Fig. 7. Separation Logic semantics

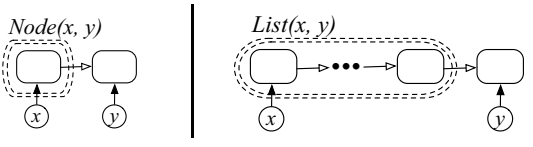


Fig. 8. Graph representation of the $Node(x, y)$ and $List(x, y)$ predicates

$(x'_1, \ldots, x'_n)$ to implicitly denote existentially quantified variables that occur in $\Pi|\Sigma$. The pure part Π is a conjunction of pure predicates which states facts about the stack variables and existential variables (e.g., $x = \mathsf{nil}$). The spatial part Σ is the $*$ conjunction of spatial predicates, i.e., related to heap facts. In separation logic, the formula $S_1 * S_2$ holds in a heap that can be split into two disjoint parts, one of them described exclusively by S_1 and the other described exclusively by S_2.

In symbolic heaps, memory locations are either pointed directly by program variables (e.g., v) or existential variables (e.g., v'), or they are abstracted by predicates. Predicates are abstractions for the graph-like structure of a set of memory locations. For example, the predicate $Node(x, y)$, in Fig. 8, abstracts a single memory location pointed by variable x, while the predicate $List(x, y)$ abstracts a set of an unbound number of memory locations, where each location is linked to another location of the set by the *next* field.

A predicate $p(\vec{e})$ has at least one parameter, from its parameter set, that is the entry point for reaching every memory location that the predicate abstracts. We denote this kind of parameter as *entry* parameters. Also, there is a subset of parameters that correspond to the exit points of the memory region abstracted by the predicate. These parameters denote variables pointing to memory locations that are outside the predicate but the predicate has memory locations with links to these *outsider* locations. In Fig. 8 we can observe that the predicate $List(x, y)$

$$
\begin{array}{lll}
H ::= v \mid v.P & & (\textit{heap path}) \\
P ::= f \mid f.P \mid C_A^*.P & & (\textit{subpath}) \\
C ::= f \mid f \text{ “|” } C & & (\text{choice})
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{S}[\![v]\!]_{s,h,l} = \{l'\} \quad \mathcal{S}[\![v.P]\!]_{s,h,l} = \mathcal{S}[\![P]\!]_{s,h,l'} & & \text{where } l' = s(v) \\
\mathcal{S}[\![f]\!]_{s,h,l} = \{l'\} \quad \mathcal{S}[\![f.P]\!]_{s,h,l} = \mathcal{S}[\![P]\!]_{s,h,l'} & & \text{where } l' = h(l,f) \\
\mathcal{S}[\![C^*.P]\!]_{s,h,l} = \mathcal{S}[\![f_1.C^*.P]\!]_{s,h,l} \cup ... \cup \mathcal{S}[\![f_n.C^*.P]\!]_{s,h,l} \cup \mathcal{S}[\![P]\!]_{s,h,l} & & \text{where } C = f_1|...|f_n
\end{array}
$$

Fig. 9. *Heap Path* syntax and semantics

has one *entry* parameter x and one *exit* parameter y. Users of StarTM are required to indicate which parameters of a predicate are *entry* or *exit* by prefixing them with the unary operators + and −, denoting *entry* and *exit* respectively. In the definition of our analysis we can query if a parameter a of a predicate p is of *entry* or *exit* type with the $\delta_p^+(a)$ or $\delta_p^-(a)$ operators respectively. For the special case of predicate ($\mapsto$), we always consider that the variable on its left side is an *entry* parameter and the variables on its right side are *exit* parameters.

4.2 Heap Paths

We are going to represent a memory location as a sequence of fields, starting from a program variable. If we successively dereference the field labels that appear in the sequence, we reach the memory location denoted by the sequence. We call these sequences of field labels, prefixed by a variable name, a *heap path*. For instance, the path $x.left.right$, denotes the memory location that is reachable by dereferencing the field *left* of the location pointed by variable x, and by dereferencing the field *right* of the location represented by $x.left$.

We can also represent sequences of field dereferences in a *heap path* by using the Kleene star ($*$) and choice ($|$) operators. For instance, the path $x.(left \mid right)^*$ denotes a memory location that can be reached by starting on variable x and then dereferencing either the *left* or *right* field on each visited memory location.

The syntax of *heap paths* is depicted in Fig. 9 and corresponds to a very restrictive subset of the regular expressions syntax. A *heap path* always starts with a variable name (v) followed by sequences of field labels (f), repeating subpath expressions under a Kleene operator (C^*), and choices of field labels (C). We syntactically restrict *heap paths*, with respect to regular expressions, by only allowing choices of field labels guarded by a Kleene operator, and repetitions of choices of single field labels (not sequences). For instance, the path $x.(left \mid right^*)$ is not a valid *heap path* expression.

Each repeating subpath is always associated with a label. This is used to identify the subpath guarded by the star and we can rewrite $C_A^*.P$ as $A.P$ where $A = C^*$. As we shall see later, this label will be used to identify subpath expressions that denote the same concrete path in the heap. We may also denote

$$
\begin{array}{lcl}
\Phi(x \mapsto [f_1 : y, \ldots, f_n : z], x, y) & = & x.f_1 \\
\Phi(p(\vec{i}, \vec{o}), x, y) & = & hp \ \text{ where } x \in \vec{i} \wedge \delta_p^+(x) \wedge y \in \vec{o} \wedge \delta_p^-(y) \wedge hp = \Gamma(p, x, y) \\
\Phi(S * S', x, y) & = & \mathsf{concat}(\Phi(S, x, z), \Phi(S', z, y)) \\
 & & \text{where exists path from } x \text{ to } z \text{ in } S \text{ and from } z \text{ to } y \text{ in } S' \\
\mathsf{concat}(x.P, z.P') = x.P.P' & &
\end{array}
$$

Fig. 10. Rules for transforming a symbolic heap into a heap path

the repetition sequence with a bar on top of the star, e.g., $x.C_A^{\bar{*}}$. This will be used to distinguish between different interpretations, of *heap path* expressions contained in read- and write-sets.

We now define the semantics of *heap paths* with relation to concrete stacks and heaps through function $\mathcal{S}[\![H]\!]_{s,h,l}$ in Fig. 9. According to this definition a *heap path* expression denotes the set of all memory locations that can be reachable by following it in a concrete memory, $\mathcal{S}[\![H]\!] \subseteq \mathsf{Locations}$. Abstract read- or write-sets are sets of *heap paths*. We write HPaths for the set of all *heap path* expressions.

In the following developments we interpret read-sets, may write-sets, and must write-sets in three different ways. For read-sets we always consider the saturation of the read-set with the denotations of all prefixes of its heap-paths. For *must* write-sets we consider one under-approximation where a heap-path H represents exactly one location in the set $\mathcal{S}[\![H]\!]$. For *may* write-sets we consider the over-approximation by saturating the set with the expansion of the $\bar{*}$ repetition annotation. For instance, a heap path expression $x.C^{\bar{*}}.f$ in a may *write-set*, denotes write operations on all fields f for all locations of the set $\mathcal{S}[\![x.C^{\bar{*}}]\!]$.

4.3 From Symbolic Heaps to Heap Paths

During the symbolic execution, we generate *heap paths* based on the information given by the symbolic heap. Recall that the only information given by the user to the verification tool is a description of the state at the beginning of the transaction using a symbolic heap, everything else is inferred.

Given a memory location l pointed by some variable x, if there is a path in the symbolic heap from some other variable s, where $s \in \mathsf{SVars}$, to variable x, then we can generate a *heap path* that represents the path from the shared variable s to the memory location l. Moreover, the computation of a *heap path* from the symbolic heap requires a transformation function that given a predicate and its arguments returns a *heap path*. In this case, the separation conjunction operator $(*)$ corresponds to the concatenation in the *heap path*. See Fig. 10 for the whole set of transformation rules. Function $\mathsf{concat}(x.P, z.P')$ concatenates the path described by P' to the *heap path* $x.P$. Note that this concatenation is sound, given the pre-condition that $x.P$ represents the same memory location as variable z, which is true in the case above.

The rule for transforming a predicate $p(\vec{i}, \vec{o})$ into a *heap path* relies on a function Γ that returns a *heap path* given a predicate and a pair of variables. A predicate definition can be transformed into a DFA (Deterministic Finite Automaton) where states correspond to predicates and transitions' labels correspond to fields. Then, we can generate a *heap path* expression, from the automaton, using well know automata to regular expressions transformation techniques. Consider the example of a *heap path* generated for the list segment predicate:

Example 1 (Heap Path of the List Segment Predicate).

$$List(x, y) \Leftrightarrow x \neq y \wedge \big(x \mapsto [next : y] \vee \exists z'.\, x \mapsto [next : z'] * List(z', y)\big)$$

Given the *List* predicate definition, the *heap path* that represents the memory location pointed by y reachable from x is:

$$\Gamma(List(x, y), x, y) = x\,.\,next_A^+$$

We abbreviate repeating sequences with at least one field label using symbol + (e.g. $next^+$). The label A is fresh in the context of the symbolic state where the *heap path* is computed. Notice that *heap path* expressions containing repetitions and choices are only generated when transforming recursive predicates into *heap paths*.

5 Symbolic Execution

Next, we define the symbolic execution for the core language presented in Section 3 taking inspiration from [8]. In our case, the symbolic execution defines the effect of statements on symbolic states composed by a symbolic heap, a path map, and a read- and write-set. We represent a symbolic state as: $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W} \rangle \in (\mathsf{SHeaps} \times (\mathsf{Vars} \rightharpoonup \mathsf{HPaths}) \times \mathsf{Rs} \times \mathsf{Ws})$ where SHeaps is the set of all symbolic heaps, $(\mathsf{Vars} \rightharpoonup \mathsf{HPaths})$ is the map between program variables and *heap path* expressions, Rs is the set of all read-sets, and Ws is the set of pairs of all *may* and *must* write-sets. We write $\mathsf{SStates}$ for denoting the set of all symbolic states.

The path map $\mathcal{M}$ is a map that associates variables to *heap path* expressions. In each state of the symbolic execution, a variable x in this map is associated with a *heap path* expression that represents the memory location pointed by x. The purpose of this map is to keep a *heap path* expression less abstract than the one that we can capture from the symbolic heap. For instance, in the map, we may have the information that we only accessed the *left* field of each node of a tree, but from the symbolic heap we get the information that we accessed the *left* or *right* fields in each node. The symbolic execution will always maintain the invariant $S_p \subseteq G_p$ where S_p is the *heap path* in the path map and G_p is the *heap path* from the symbolic heap, for a variable x. The subset relation means that all paths described by S_p are described by G_p.

Each transactional method is annotated with the **@Atomic** annotation describing the initial symbolic heaps for that transaction. The symbolic execution will analyze only transactional methods and all methods present in the invocation

tree that occurs inside their body. In the beginning of the analysis we have the specification of the symbolic heaps for each transactional method. An empty path map and empty read- and write-sets are associated to each initial symbolic heap, thus creating a set of initial symbolic states for each transactional method. The complete information for each method is composed by:

- the initial symbolic states, which can be given by the programmer or be computed by the analysis;
- the final symbolic states resulting from the method's execution. These final symbolic states are computed by the analysis and, in the special case of the transactional methods, are the final result of the analysis.

For each method, given one initial symbolic state, the analysis may produce more than one symbolic states. The symbolic execution is defined by function exec that yields a set of symbolic states or an error ($\top$), given a method body (from Stmt) and an initial symbolic state (from $\mathsf{SStates}$):

$$\mathsf{exec} : \mathsf{Stmt} \times \mathsf{SStates} \rightarrow \mathcal{P}(\mathsf{SStates}) \cup \{\top\}$$

To support inter-procedural analysis we also need the auxiliary function spec, that given a method signature ($\mathit{fun}(\vec{x}) \in \mathsf{Sig}$), yields a mapping from symbolic heaps to sets of symbolic states: $\mathsf{SHeaps} \rightarrow \mathcal{P}(\mathsf{SStates})$.

$$\mathsf{spec} : \mathsf{Sig} \rightarrow (\mathsf{SHeaps} \rightarrow \mathcal{P}(\mathsf{SStates}))$$

For non-transactional methods, called inside transactions, the initial symbolic state is computed in the course of the symbolic execution, which is inferred from the symbolic state of the calling context. Recursive functions are currently not supported by our analysis technique.

5.1 Past Symbolic Heap

In our analysis we need a special kind of predicates, which we call *past predicates*, and are denoted as $\widehat{p}(\vec{e})$ or $x \widehat{\mapsto} [\rho]$. The past symbolic heap is composed by predicates and past predicates. The latter ones have an important role in the correctness for computing *heap paths*. *Heap paths* must always be computed with respect to the initial snapshot of memory, which is shared between transactions, and corresponds to the initial symbolic heap. Otherwise we may fail to detect some shared memory access due to some memory privatization pattern. We illustrate this problem by means of an example:

Example 2. Given an initial symbolic heap, where $x \in \mathsf{SVars}$ is a shared variable:

$$\{\}|\ \mathit{List}(x, y) * y \mapsto [\mathit{next} : z] * z \mapsto \mathsf{nil}$$

The *heap paths* representing the locations pointed by each variable are:

$$x \equiv x \qquad y \equiv x.(\mathit{next})^{+}_{A} \qquad z \equiv x.(\mathit{next})^{+}_{A}.\,\mathit{next}$$

If we update the location pointed by y by assigning its *next* field to nil we get

$$\{\}|\, List(x, y) * y \mapsto [next : \mathsf{nil}] * z \mapsto \mathsf{nil}$$

After the update, the *heap paths* representing the locations pointed by x and y remain the same. However, z is no longer reachable from a shared variable, and hence, we have lost the information that in the context of a transaction, z is still a shared memory location subject to concurrent modifications.

This example shows that the *heap path* representing a memory location, that is reachable by a shared variable in the beginning of the transaction, must not be changed by the updates in the structure of the heap. So, in order to compute the correct *heap path* we need to use a "past view" of the current symbolic heap. To get the past view we need *past* predicates, which are added to the symbolic heap whenever an update is made to the structure of the heap. In the case of the previous example, the result of updating variable y would give the following symbolic heap:

$$\{\}|\, List(x, y) * y \mapsto [next : \mathsf{nil}] * y \widehat{\mapsto} [next : z] * z \mapsto \mathsf{nil}$$

The *past* predicate $y \widehat{\mapsto} [next : z]$ denotes that there was a *link* between variable y and z in the initial symbolic heap. Now, if there is a read access to a field of the memory location pointed by variable z, we compute the *heap path* of this location in the past view of the symbolic heap. We define a function that given a symbolic heap returns the past view of such symbolic heap:

Definition 2 (Past Symbolic Heap). *Let* $\mathsf{Past}(H)$ *be the set of past predicates in* H*, and* $\mathsf{NPast}(\Pi|\Sigma) = \{S \mid \Sigma = S * \Sigma' \wedge \neg\, \mathsf{hasPast}_{\Pi|\Sigma}(S)\}$*. Then we define the past symbolic heap by*

$$\mathsf{PSH}(\Pi|\Sigma) \triangleq \Pi|\, \circledast_{S \in \mathsf{NPast}(\Pi|\Sigma)}\, S * \circledast_{\widehat{S} \in \mathsf{Past}(\Pi|\Sigma)} \widehat{S}$$

This function makes use of the hasPast function to assert if there is already a *past predicate*, in the symbolic heap, with the same *entry* parameters. We define hasPast as:

Definition 3 (Has Past).

$$\begin{aligned} &\mathsf{hasPast}_{\mathcal{H}}(x \mapsto [\rho]) \Leftrightarrow \mathcal{H} \vdash x \widehat{\mapsto} [\rho] * true \\ &\mathsf{hasPast}_{\mathcal{H}}(p(\vec{i}, \vec{o})) \quad \Leftrightarrow \forall i \in \vec{i} : \delta_p^+(i) \wedge \exists i \in \vec{i} : \mathcal{H} \vdash \widehat{p}(\ldots, i, \ldots) * true \end{aligned}$$

The result of the past heap function applied to the previous example is:

$$\begin{aligned} &\mathsf{PSH}(\{\}|\, List(x, y) * y \mapsto [next : \mathsf{nil}] * y \widehat{\mapsto} [next : z] * z \mapsto \mathsf{nil}) \\ &\quad \triangleq \{\}|\, List(x, y) * y \mapsto [next : z] * z \mapsto \mathsf{nil} \end{aligned}$$

Which corresponds to the initial symbolic heap of Example 2. Thus we can calculate correctly the *heap paths* of the locations pointed by x, y and z.

We also define a function $\mathsf{PastOf}_{\mathcal{H}}(x \mapsto [\rho])$ that if the symbolic heap $\mathcal{H}$ does not contain a past points-to predicate for a points-to predicate $x \mapsto [\rho]$, it creates a new past predicate $x \widehat{\mapsto} [\rho]$.

$$\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, S\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R}', \mathcal{W}'\rangle \quad \vee \quad \langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, S\rangle \Longrightarrow \top$$

$$I(e) ::= e.f := x \mid x := e.f$$

$$\frac{\mathcal{H} \vdash y = \mathsf{nil}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, I(y)\rangle \Longrightarrow \top}(\textsc{Heap Error})$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := e\rangle \Longrightarrow \langle x = e[x'/x] \wedge \mathcal{H}[x'/x], \mathcal{M}[x \mapsto \mathcal{M}(e)], \mathcal{R}, \mathcal{W}\rangle}(\textsc{Assign})$$

$$\frac{\begin{array}{c} p = \mathsf{GenP}(\mathsf{PSH}(\mathcal{H}), \mathcal{M}, y) \qquad \mathcal{M}' = \mathsf{uMap}(\mathcal{M}, \mathcal{H}, y, p)[x \mapsto p.f] \\ \mathcal{H}' = x = z[x'/x] \wedge \mathcal{H}[x'/x] \qquad x' \text{ is fresh} \end{array}}{\langle \mathcal{H} * y \mapsto [f : z], \mathcal{M}, \mathcal{R}, \mathcal{W}, x := y.f\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R} \cup \{p.f\}, \mathcal{W}\rangle}(\textsc{Heap Read})$$

$$\frac{\begin{array}{c} p = \mathsf{GenP}(\mathsf{PSH}(\mathcal{H} * x \mapsto [f : z]), \mathcal{M}, x) \qquad \mathcal{M}' = \mathsf{uMap}(\mathcal{M}, \mathcal{H}, x, p) \\ \mathcal{H}' = \mathcal{H} * x \mapsto [f : e] * \mathsf{PastOf}_{\mathcal{H}}(x \mapsto [f : z]) \end{array}}{\langle \mathcal{H} * x \mapsto [f : z], \mathcal{M}, \mathcal{R}, \mathcal{W}, x.f := e\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R}, \mathcal{W} \uplus \{p.f\}\rangle}(\textsc{Heap Write})$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := \mathsf{new}\rangle \Longrightarrow \langle \mathcal{H}[x'/x] * x \mapsto [], \mathcal{M}[x \mapsto \epsilon], \mathcal{R}, \mathcal{W}\rangle}(\textsc{Allocation})$$

$$\frac{}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, \mathsf{return}\, e\rangle \Longrightarrow \langle ret = e \wedge \mathcal{H}, \mathcal{M}[ret \mapsto \mathcal{M}(e)], \mathcal{R}, \mathcal{W}\rangle}\ (\textsc{Return})$$

$$\frac{\begin{array}{c} \langle \mathcal{H}'', \mathcal{M}', \mathcal{R}', \mathcal{W}'\rangle \in \mathsf{spec}(fun(\vec{z}))(\mathcal{H}') \quad \mathcal{H} \vdash \mathcal{H}'[\vec{y}/\vec{z}] * Q \quad \mathcal{H}''' = Q * \mathcal{H}''[\vec{y}/\vec{z}] \\ \mathcal{R}'' = \mathcal{R}'[\vec{y}/\vec{z}] \qquad \mathcal{W}'' = \mathcal{W}'[\vec{y}/\vec{z}] \qquad \mathcal{M}'' = \mathsf{uAMap}(\mathcal{R}'' \cup \mathcal{W}'', \mathcal{M}, \mathcal{H}''') \\ r.P' = \mathcal{M}'(ret) \qquad \mathcal{M}''' = \mathcal{M}''[x \mapsto \mathsf{GenP}(\mathsf{PSH}(\mathcal{H}'''), \mathcal{M}'', r).P'] \\ \mathcal{R}''' = \mathcal{R} \cup \{\mathcal{M}'''(v).P \mid v.P \in \mathcal{R}''\} \qquad \mathcal{W}''' = \mathcal{W} \uplus \{\mathcal{M}'''(v).P \mid v.P \in \mathcal{W}''\} \end{array}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := fun(\vec{y})\rangle \Longrightarrow \langle x = ret \wedge \mathcal{H}''', \mathcal{M}''', \mathcal{R}''', \mathcal{W}'''\rangle}(\textsc{FCall})$$

$$\mathsf{alias}_{\mathcal{H}}(x) \triangleq \{y \mid \mathcal{H} \vdash x = y\} \cup \{x\}$$

$$\mathsf{uMap}(\mathcal{M}, \mathcal{H}, x, p) \triangleq \{v \mapsto s \mid v \mapsto s \in \mathcal{M} \wedge v \notin \mathsf{alias}_{\mathcal{H}}(x)\} \cup \{a \mapsto p \mid a \in \mathsf{alias}_{\mathcal{H}}(x)\}$$

$$\mathsf{uAMap}(V, \mathcal{M}, \mathcal{H}) \triangleq \{s \mid v.P \in V \wedge p = \mathsf{GenP}(\mathsf{PSH}(\mathcal{H}), \mathcal{M}, v) \wedge s \in \mathsf{uMap}(\mathcal{M}, \mathcal{H}, v, p)\}$$

Fig. 11. Operational Symbolic Execution Rules

Definition 4 (Generate Past Predicate).

$$\mathsf{PastOf}_{\mathcal{H}}(x \mapsto [\rho]) \triangleq \begin{cases} \mathsf{emp} & \textit{if } \mathsf{hasPast}_{\mathcal{H}}(x \mapsto [\rho]) \\ x \overset{\frown}{\mapsto} [\rho] & \textit{otherwise} \end{cases}$$

5.2 Symbolic Execution Rules

The symbolic execution is defined by the rules shown in Fig. 11.

The rule Assign, when executed in a state $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}\rangle$ adds the information that in the resulting state, x is equal to e. As in standard Hoare/Floyd style assignment, all the occurrences of x, in $\mathcal{H}$ and e, are replaced by a fresh existential

quantified variable x'. We also compute a new path map where we associate variable x with the *heap path* of expression e. If e is null then we associate variable x with empty ϵ. The read- and write-set are not changed because there are no changes in the heap.

The HEAP READ rule adds an equality, to the resulting state, between x and the content of the field f of the location pointed by y. Every time we access the heap, for reading or writing, we compute a new path map. In this case we generate a *heap path* for variable y using the symbolic heap and the current path map. Note that the *heap path* generated is computed in the past symbolic heap as described in Section 5.1. This operation, denoted as GenP, is also responsible for abstracting the representation of *heap paths*, we will describe it in detail in Section 5.4. Given the new computed *heap path* p we compute a new path map by associating path p with variable y, and all its aliases. We use function uMap to perform these operations. Then we associate variable x with the result of the concatenation of path p, which represents the memory location pointed by y, with field f. Finally, we add to the read-set the memory access represented by the *heap path* p and the field f.

The HEAP WRITE rule denotes an update to the value of field f in the location pointed by x. Variable x is associated with the generated *heap path* p ($\mathsf{uMap}(\mathcal{M}, \mathcal{H}, x, p)$) in a new path map. The symbolic heap is extended with a past predicate representing the link between variable x and the record $[f : z]$ that just ceased to exist. The resulting write-set is extended with the field access $\{p.f\}$ ($\mathcal{W} \Cup \{p.f\}$). The operation $\mathcal{W} \Cup \{p.f\}$, denotes the adding of $\{p.f\}$ to both components of the write set $\mathcal{W}$, to the *may* write-set $\mathcal{W}^{>}$ and to the *must* write-set $\mathcal{W}^{<}$. While adding an *heap path* access $p.f$ to the *must* write-set $\mathcal{W}^{<}$ is straightforward, adding $p.f$ to the *may* write-set $\mathcal{W}^{>}$ is a bit more involved. If $\mathcal{W}^{>}$ already contains $p.f$, then we replace all repeating sequences in p, by repeating sequences of the kind $\bar{*}$. For instance, in the previous example, if $p.f = x\,.\,next^{*}_{A}\,.\,next$ is already in $\mathcal{W}^{>}$, the may write-set after adding $p.f$ contains $x\,.\,next^{\bar{*}}_{A}\,.\,next$ instead.

When a new memory location is allocated, rule ALLOCATION, and is assigned to variable x we update the path map entry for variable x with *empty* (ϵ).

In the FCALL rule, the function spec is used to get the symbolic state $\langle \mathcal{H}'', \mathcal{M}', \mathcal{R}', \mathcal{W}' \rangle$ which corresponds to one of the final states of the symbolic execution of a function *fun*. The read- and write-set are composed by *heap path* expressions, where each expression $v.P$ represents a memory location where variable v is the root of the path. This variable is a root variable in the context of function *fun* but in the context of the function that is being analyzed where *fun* was invoked, variable v might point to a memory location that is represented by a *heap path* expression $v'.P'$ where $v' \neq v$. This means that a memory location that is represented by the expression $v.P$ in the context of *fun*, is represented by the expression $v'.P'.P$ in the context of the calling site of *fun* where $v'.P'$ is the expression that represent the memory location pointed by v in the context of the calling site. We need to update all *heap path* expressions of all variables that are in the returned read- ($\mathcal{R}'$) and write-set ($\mathcal{W}'$). We use the uAMap function

to iterate over all variables and generate a new *heap path* expression and update the path map accordingly. The return value of function *fun* is assigned to variable x and therefore we update the path map entry for variable x with the *heap path* expression that represents memory location pointed by the special return variable *ret* in the context of the calling site. In the last step, we merge the read- and write-sets using the updated path map $\mathcal{M}'''$ by concatenating the *heap path* $\mathcal{M}'''(v)$ with the remaining path returned from the read- ($\mathcal{R}'$) or write-set ($\mathcal{W}'$). The final symbolic heap $\mathcal{H}'''$ is computed in the typical way for inter-procedural analysis using separation logic that is by combining the frame of the function call (in this case Q)[3], and the postcondition of the spec $\mathcal{H}''$ [9].

Since we are not aiming at verifying execution errors, we silently ignore the symbolic error states ($\top$) produced by Heap Error rule in our analysis.

5.3 Rearrangement Rules

The symbolic execution rules manipulate object's fields. When these are hidden inside abstract predicates both Heap Read and Heap Write rules require the analyzer to expose the fields they are operating on. This is done by the function rearr defined as:

Definition 5 (Rearrangement).

$$\mathsf{rearr}(\mathcal{H}, x.f) \triangleq \{\mathcal{H}' * x \mapsto [f : y] \mid \mathcal{H} \vdash \mathcal{H}' * x \mapsto [f : y]\}$$

5.4 Fixed Point Computation and Abstraction

Following the spirit of abstract interpretation [6] and the jStar work [9] to ensure termination of symbolic execution, and to automatically compute loop invariants, we apply abstraction on sets of symbolic states. Typically, in separation logic based program analyses, abstraction is done by rewriting rules, also called abstraction rules which implement the function $\mathsf{abs} : \mathsf{SHeaps} \rightarrow \mathsf{SHeaps}$. For each analyzed statement we apply abstraction after applying the execution rules. The abstraction rules accepted by StarTM have the form:

$$\frac{premises}{\mathcal{H} \vdash \mathsf{emp} \rightsquigarrow \mathcal{H}' \vdash \mathsf{emp}} (\text{Abstraction Rule})$$

This rewrite is sound if the symbolic heap $\mathcal{H}$ implies the symbolic heap $\mathcal{H}'$. An example of some abstraction rules, for the $List(x, y)$ predicate, is shown in Fig. 2.

The *heap path* expressions that are stored in the path map ($\mathcal{M}$) need also to be abstracted because otherwise we would get expressions with infinite sequences of fields. Since the symbolic heap is abstracted we can use it to compute an abstract *heap path* expression. The abstraction procedure is done by the $\mathsf{GenP}(\mathcal{H}, \mathcal{M}, v)$ function. This function receives a symbolic heap $\mathcal{H}$, a path map $\mathcal{M}$, and a

[3] The frame of a call is the part of the calling heap which is not related with the precondition of the callee.

$$
\begin{aligned}
\mathsf{compress}(f_1.f_2) &= (f_1)^+_A && \text{if } f_1 = f_2 \quad \text{where } A \text{ is fresh}\\
\mathsf{compress}(f_1.f_2) &= (f_1|f_2)^+_A && \text{if } f_1 \neq f_2 \quad \text{where } A \text{ is fresh}\\
\mathsf{compress}((\mathcal{C})^+_C.f_1) &= (\mathcal{C})^+_C && \text{if } f_1 \in \mathcal{C}\\
\mathsf{compress}((\mathcal{C})^+_C.f_1) &= (\mathcal{C}|f_1)^+_C && \text{if } f_1 \notin \mathcal{C}\\
\mathsf{compress}(f_1.f_2.P) &= \mathsf{compress}(\mathsf{compress}(f_1.f_2).P)
\end{aligned}
$$

Fig. 12. Compress abstraction function

variable v for which will be computed the *heap path* representing the memory location pointed by such variable.

The *heap path* stored in the path map $\mathcal{M}$ for variable v will be denoted as S, and the *heap path* computed from the symbolic heap will be denoted as G. The analysis will always ensure the invariant that $S \subseteq G$. This subset relation means that all paths described by S are also described by G.

The result of this function is a *heap path*, denoted as E which satisfies the following invariant: $S \subseteq E \subseteq G$. Since the symbolic heap is proven to converge into a fixed point, the *heap path* E will also converge into a fixed point because it is a subset of G.

The procedure to compute the path E is based on a pattern matching approach. Taking G as the most abstract path we generate a pattern from it that must match in S. This pattern is generated by taking G and substituting all its repeating sequences with wildcards. For instance, if $G = x.(\mathit{left} \mid \mathit{right})^+_A.\mathit{right}$ then the pattern would be $Pt = x.\alpha.\mathit{right}$ where α is a wildcard. We also denote α_G as the subpath in G that is associated to the wildcard α, and in this case, $\alpha_G = (\mathit{left} \mid \mathit{right})^+_A$.

We take this pattern and try to apply it to S and check which subpath expression of S matches the wildcard. For instance, if $S = x.\mathit{left}.\mathit{left}.\mathit{right}$, then the wildcard α of pattern $Pt = x.\alpha.\mathit{right}$ will match $\mathit{left}.\mathit{left}$ denoted as α_S. The pattern can only be matched successfully if the wildcard in S (α_S) and the wildcard in G (α_G) satisfy the following invariant: $\alpha_S \subseteq \alpha_G$, which is the case in our example.

Now we apply an abstraction operation over the wildcard to generate a more abstract subpath. We denote this operation as compress and is defined in Fig. 12. The result of applying the abstraction function to wildcard α_S is $\mathsf{compress}(\alpha_S) = \mathit{left}^+_B$. Notice that the abstracted subpath satisfies the invariant $\alpha_S \subseteq \mathsf{compress}(\alpha_S) \subseteq \alpha_G$. Finally, we substitute the wildcards in the pattern for the computed abstract subpath expressions. In our example we get the final expression $E = x.\,\mathit{left}^+_B.\mathit{right}$ which is a subset of G.

5.5 Write-Skew Detection

The result of the symbolic execution is a set of symbolic states $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W} \rangle$ for each transactional method. In this section, we define the *write-skew* test, which

is based on the abstract read- and write-set $(\mathcal{R}, \mathcal{W})$ and on the satisfiability of the condition of Definition 1 (see example in Fig. 3).

Recall that the interpretation of read-sets contain all prefixes of its heap paths. Hence, to compute the satisfiability of the *write-skew condition* we must compute the set of prefixes of the heap-paths in both read-sets. We define $\mathsf{prefix}(x.P)$ for a heap path expression $x.P$ as follows:

$$\mathsf{prefix}(P.f) \triangleq \{P.f\} \cup \mathsf{prefix}(P) \qquad \mathsf{prefix}(P.C_A^*) \triangleq \{P.C_A^*\} \cup \mathsf{prefix}(P)$$
$$\mathsf{prefix}(x.f) \triangleq \{x.f\} \qquad \mathsf{prefix}(x.C_A^*) \triangleq \{x.C_A^*\}$$

and define it for sets of heap paths $\mathsf{prefix}(\mathcal{R})$ as

$$\mathsf{prefix}(\mathcal{R}) \triangleq \bigcup_{p \in \mathcal{R}} \mathsf{prefix}(p).$$

For instance, the prefixes of the read-set $\mathcal{R} = \{this.head.(next)_A^*.next\}$ are:

$$\mathsf{prefix}(\mathcal{R}) = \{this.head,\ this.head.A,\ this.head.A.next\}$$

For the sake of simplicity, we denote repeating sequences by their unique label. Given the sets, $\mathcal{R}_1^\star = \mathsf{prefix}(\mathcal{R}_1)$, $\mathcal{R}_2^\star = \mathsf{prefix}(\mathcal{R}_2)$, $\mathcal{W}_1^<$, $\mathcal{W}_1^>$, $\mathcal{W}_2^<$, and $\mathcal{W}_2^>$, the *write-skew* condition is the following:

$$\mathcal{R}_1^\star \cap \mathcal{W}_2^> \neq \emptyset \quad \wedge \quad \mathcal{W}_1^> \cap \mathcal{R}_2^\star \neq \emptyset \quad \wedge \quad \mathcal{W}_1^< \cap \mathcal{W}_2^< = \emptyset$$

From this condition we generate a set of (in)equations, on the labels of repeating sequences, necessary to reach satisfiability. For instance, given the sets:

$$\mathcal{R}^\star = \{this.head,\ this.head.A,\ this.head.A.next,\ this.head.A.next.next\}$$
$$\mathcal{W}^> = \{this.head.B.next\}$$

The condition $\mathcal{R}^\star \cap \mathcal{W}^> \neq \emptyset$ is satisfied if there is a possible instantiation of A and B such that:

$$B.next \leq A \quad \vee \quad B = A \quad \vee \quad B = A.next$$

In inequation $B.next \leq A$, the operator $\leq$ denotes prefixing, in this case that $B.next$ is a prefix of A. After generating the (in)equation system on labels (A, B) needed to satisfy the *write-skew* condition, we use an SMT solver to check their satisfiability. The consequence of that result is that a *write-skew* may occur between the two transactions being analyzed. Notice that when comparing read- and write-sets we make the correspondence between concrete paths in the heap through the unique labels of repeating sequences.

5.6 Soundness

Our approach is sound for the detection of the *write-skew* anomaly between pairs of transactions. We argue that, by analyzing the satisfiability test described in

section 5.5, if no *write-skew* anomaly is detected by our algorithm then there is no possible execution of the program that contains a *write-skew*. Our analysis computes an over-approximation of the concrete read- and write-sets (the may write-set), and also an under-approximation of the concrete write-set (the must write-set), for all possible executions of the program.

The question then remains whether an occurrence of a write-skew condition at runtime is captured by our test. To see this, let's assume that $\mathcal{R}_1^c$, $\mathcal{W}_1^c$, $\mathcal{R}_2^c$, $\mathcal{W}_2^c$ are concrete, exact read- and write- sets for transactions T_1 and T_2. Notice that a write-skew condition occurs between T_1 and T_2 if

$$\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset$$

Our analysis computes abstract over-approximations of read-sets ($\mathcal{R}_1$ and $\mathcal{R}_2$), write-sets ($\mathcal{W}_1^{>}$ and $\mathcal{W}_2^{>}$), and under-approximation of write-sets ($\mathcal{W}_1^{<}$ and $\mathcal{W}_2^{<}$) related to the concrete read- and write-sets as follows:

$$\mathcal{R}_1^c \subseteq \mathcal{R}_1, \quad \mathcal{R}_2^c \subseteq \mathcal{R}_2, \quad \mathcal{W}_1^c \subseteq \mathcal{W}_1^{>}, \quad \mathcal{W}_2^c \subseteq \mathcal{W}_2^{>}, \quad \mathcal{W}_1^{<} \subseteq \mathcal{W}_1^c, \quad \mathcal{W}_2^{<} \subseteq \mathcal{W}_2^c$$

These set relations allow us to prove that the condition on abstract sets is implied by the condition on concrete sets:

$$\begin{aligned}(\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset) \Rightarrow \\ (\mathcal{R}_1 \cap \mathcal{W}_2^{>} \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{>} \cap \mathcal{R}_2 \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{<} \cap \mathcal{W}_2^{<} = \emptyset)\end{aligned}$$

Hence we can conclude that if a real write-skew exists in an execution this will be detected by our test, and this, as consequence, makes our method sound. The implication above also shows that our method may present false positives: it may detect a write-skew that will never occur at runtime. This is a classical unavoidable effect of conservative methods based on abstract interpretation.

6 Experimental Results

StarTM is a prototype implementation of our static analysis applied to Java byte code, using the Soot toolkit [20] and the CVC3 SMT solver [1]. We applied StarTM to three STM benchmarks: an ordered linked list, a binary search tree, and the Intruder test program of the STAMP benchmark. In the case of the list we tested two versions: the unsafe version called List and the safe version called List Safe. The List Safe version has an additional update in the remove method as discussed in Section 2.

Table 1 shows the detailed results of our verification for each transactional method of the examples above. The results were obtained in a Intel Dual-Core i5 650 computer, with 4 GB of RAM. We show the time (in seconds) taken by StarTM to verify each example, the number of lines of code, and the number of states produced during the analysis. The last column in the table shows the pairs of transactions that may actually trigger a *write-skew* anomaly.

Table 1. StarTM applied to STM benchmarks

Bench.	Method	Time	LOC	States	*Write-Skews*
List	add	5	16	2	(add, remove) (remove, remove)
	remove		14	2	
	contains		11	1	
	revert		11	4	
List Safe	add	6	16	2	-
	remove		15	2	
	contains		11	1	
Tree	treeAdd	11	21	3	-
	treeContains		15	2	
Intruder	atomicGetPacket	24	9	2	(atomicProcess, atomicGetComplete)
	atomicProcess		173	7	
	atomicGetComplete		15	2	

The expected results for the two versions of the linked list benchmark were confirmed by our tool. The tool detects the existence of two *write-skew* anomalies, in the unsafe version of the linked list, resulting from the concurrent execution of the add and remove methods. The safe version is proven to be completely safe when executing all transactions under SI.

In the case of the Tree benchmark, the treeAdd method performs a tree traversal and inserts a new leaf node. StarTM proves that the concurrent execution of all transactions of the Tree benchmark is safe.

StarTM detects a *write-skew* anomaly in the Intruder example, which is triggered by the concurrent execution of atomicProcess and atomicGetComplete transactions. This happens when transaction atomicProcess pushes an element into a stack and transaction atomicGetComplete pops an element from the same stack, which result on writes on different parts of the memory. However, the Intruder example is not entirely analyzed, there is a small part of the code that is not analyzed due to the use of arrays and cyclic data-structures, which are not currently supported by our tool.

These promising results together with the known performance advantages [7] support the key idea of using relaxed isolation levels in transactional memory systems.

7 Related Work

Software Transactional Memory (STM) [18,11] systems commonly implement the full serializability of memory transactions to ensure the correct execution of concurrent programs. To the best of our knowledge, SI-STM [17] is the only existing implementation of a STM using snapshot isolation. This work focuses on improving the transactional processing throughput by using a snapshot isolation

algorithm. It proposes a SI safe variant of the algorithm, where anomalies are dynamically avoided by enforcing additional validation of read-write conflicts. Our approach avoids this validation by using static analysis and correcting the anomalies before executing the program.

In our work, we aim at providing the serializability semantics under snapshot isolation for STM and Distributed STM systems. This is achieved by performing a static analysis of the program and asserting that no SI anomalies will ever occur when executing a transactional application. This allows to avoid tracking read accesses in both read-only and read-write transactions, thus increasing performance throughput.

The use of snapshot isolation in databases is a common place, and there are some previous works on the detection of SI anomalies in this domain. Fekete et al. [10] developed the theory of SI anomalies detection and proposed a syntactic analysis to detect SI anomalies for the database setting. They assume applications are described in some form of pseudo-code, without conditional (*if-then-else*) and cyclic structures. The proposed analysis is informally described and applied to the database benchmark TPC-C [19] proving that its execution is safe under SI. A sequel of that work [12], describes a prototype which is able to automatically analyze database applications. Their syntactic analysis is based on the names of the columns accessed in the SQL statements that occur within the transaction.

Although targeting similar results, our work deals with different problems. The most significant one is related to the full power of general purpose languages and the use of dynamically allocated heap data structures. To tackle this problem, we use separation logic [16,8] to model operations that manipulate heap pointers. Separation logic has been the subject of research in the last few years for its use in static analysis of dynamic allocation and manipulation of memory, allowing one to reason locally about a portion of the heap. It has been proven to scale for larger programs, such as the Linux kernel [4].

The approach described in [15] has a close connection to ours. It defines an analysis to detect memory independences between statements in a program, which can be used for parallelization. They extended separation logic formulae with labels, which are used to keep track of memory regions through an execution. They can prove that two distinct program fragments use disjoint memory regions on all executions, and hence, these program fragments can be safely parallelized. In our work, we need a finer grain model of the accessed memory regions. We also need to distinguish between read and write accesses to shared and separated memory regions.

The work in [14] informally describes a similar static analysis to approximate read- and write-sets using escape graphs to model the heap structure. Our shape analysis is based on separation logic, and, as far as we understand, heap-paths give a more fine-grain representation of memory locations at a possible expense in scalability.

Some aspects of our work are inspired by jStar [9]. jStar is an automatic verification tool for Java programs, based on separation logic, that enables the

automatic verification of entire implementations of several design patterns. Although our work has some aspect in common with jStar, the properties being verified are completely different.

8 Concluding Remarks

We describe a novel and sound approach to automatically verify the absence of the *write-skew* snapshot isolation anomaly in transactional memory programs. Our approach is based on a general model for fine grain abstract representation of accesses to dynamically allocated memory locations. By using this representation, we accurately approximate the concrete read- and write-sets of memory transactions, and capture *write-skew* anomalies as a consequence of the satisfiability of an assertion based on the output of the analysis, the abstract read- and write-sets.

We present StarTM, a prototype implementation of our theoretical framework, unveiling the potential for the safe optimization of transactional memory Java programs by relaxing isolation between transactions. Our approach is not without limitations. Issues that require further developments range from the generalization of the *write-skew* condition for more than two transactions, the support for richer dynamic data structures, to the support for array data types. Together with a runtime system support for mixed isolation levels, we believe that our approach can scale up to significantly optimize real-world transactional memory systems.

Acknowledgments. We are grateful to the anonymous reviewers for several insightful comments that significantly improved the paper.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Berenson, H., Bernstein, P., Gray, J.N., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: SIGMOD 1995: Proc. of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 1–10. ACM, New York (1995)
3. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 101–112. ACM, New York (2008)
4. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 289–300. ACM, New York (2009)
5. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008: Proc. IEEE Int. Symp. on Workload Characterization (2008)

6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1977, pp. 238–252. ACM, New York (1977)
7. Dias, R.J., Loureno, J.M., Preguia, N.M.: Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In: 3rd USENIX Conference on Hot Topics in Parallelism (HotPar 2011). Usenix Association (2011)
8. Distefano, D., O'Hearn, P.W., Yang, H.: A Local Shape Analysis Based on Separation Logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
9. Distefano, D., Parkinson, M.J.: jstar: towards practical verification for Java. In: Proc. of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA 2008), pp. 213–226. ACM, New York (2008)
10. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2), 492–528 (2005)
11. Herlihy, M., Luchangco, V., Moir, M., William, N., Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC 2003: Proc. of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 92–101. ACM, New York (2003)
12. Jorwekar, S., Fekete, A., Ramamritham, K., Sudarshan, S.: Automating the detection of snapshot isolation anomalies. In: VLDB 2007: Proc. of the 33rd International Conference on Very Large Data Bases, pp. 1263–1274. VLDB Endowment, Vienna (2007)
13. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with Java STM. In: MultiProg 2010: Programmability Issues for Heterogeneous Multicores (2010)
14. Prabhu, P., Ramalingam, G., Vaswani, K.: Safe programmable speculative parallelism. In: Proc. of the 2010 ACM SIGPLAN Conf. on Prog. Language Design and Implementation, PLDI 2010, pp. 50–61. ACM, New York (2010)
15. Raza, M., Calcagno, C., Gardner, P.: Automatic Parallelization with Separation Logic. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 348–362. Springer, Heidelberg (2009)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS 2002, pp. 55–74. IEEE Computer Society, Washington, DC (2002)
17. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: TRANSACT 2006: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, Ottawa, Canada (2006)
18. Shavit, N., Touitou, D.: Software transactional memory. In: PODC 1995: Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213. ACM, New York (1995)
19. Transaction Processing Performance Council: TPC-C benchmark, revision 5.11 (2010)
20. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1999, p. 13. IBM Press (1999)

Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates

Arnab De and Deepak D'Souza

Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, India
{arnabde,deepakd}@csa.iisc.ernet.in

Abstract. The ability to perform strong updates is the main contributor to the precision of flow-sensitive pointer analysis algorithms. Traditional flow-sensitive pointer analyses cannot strongly update pointers residing in the heap. This is a severe restriction for Java programs. In this paper, we propose a new flow-sensitive pointer analysis algorithm for Java that can perform strong updates on heap-based pointers effectively. Instead of points-to graphs, we represent our points-to information as maps from access paths to sets of abstract objects. We have implemented our analysis and run it on several large Java benchmarks. The results show considerable improvement in precision over the points-to graph based flow-insensitive and flow-sensitive analyses, with reasonable running time.

1 Introduction

Pointer analysis is used to determine if a pointer may point to an abstract memory location, typically represented by an allocation site in languages like Java. A precise pointer analysis has the potential to increase the precision and scalability of client program analyses [29,17]. The precision of pointer analysis can be improved along two major dimensions: *flow-sensitivity* and *context-sensitivity*. A flow-insensitive pointer analysis [1,31] computes a single points-to information for the entire program that over-approximates the possible points-to relations at all states that the program may reach at run-time. A flow-sensitive analysis on the other hand takes the control flow structure of a program into account and produces separate points-to information at every program statement. A context-sensitive analysis aims to distinguish among invocations of the same function based on the calling contexts.

Traditionally researchers have focused on improving the scalability and precision of flow-insensitive [14,2,26,10] and context-sensitive analyses [25,34,33]. Flow-sensitive analyses were found to be expensive and gave little additional payoff in client applications like memory access optimizations in compilers [16,15,17]. However in recent years, it has been observed that several client analyses like typestate verification [8], security analysis [5], bug detection [9], and the analysis of multi-threaded programs [28], can benefit from a precise flow-sensitive pointer

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 665–687, 2012.

analysis. As a result there has been renewed interest in the area of flow-sensitive pointer analysis and the scalability of such analyses, particularly for C programs, has been greatly improved [12,22,38,21,11,18].

Most of these techniques however compute the points-to information as a *points-to graph* (or some variant of it), which as we explain below, can be a severe limitation to improvements in precision for Java programs. A node in a points-to graph can be a variable or an abstract object representing a set of dynamically allocated heap objects. Figure 1(b) shows an example points-to graph. Typically, allocation sites are used as abstract objects to represent all concrete objects allocated at that site. An edge from a variable to an abstract object denotes that the variable may point to that object. Similarly an edge from an abstract object `o1` to an abstract object `o2`, annotated with field `f`, denotes that the `f` field of object `o1` may point to the object `o2`[1]. Precision improvements of flow-sensitive pointer analyses come mostly from the ability to perform *strong updates* [21]. If the analysis can determine that an assignment statement writes to a single concrete memory location, it can *kill* the prior points-to edges of the corresponding abstract memory location. It requires the lhs of the assignment to represent a single abstract memory location *and* that abstract memory location to represent a single concrete memory location. As abstract objects generally represent multiple concrete objects, the analysis cannot perform a strong update on such objects. This situation is common in Java programs, where all indirect assignment statements (i.e. assignments whose lhs have at least one dereference) write to the heap, and hence traditional flow-sensitive algorithms cannot perform any strong updates for such assignments.

We illustrate this problem using the program fragment of Figure 1(a). The points-to graph before statement `L1` is shown in Figure 1(b), where variables `p` and `r` point to the abstract heap location `o1`, `q` points to `o3`, and field `f` of object `o1` points to the object `o2`. If the abstract object `o1` represents multiple concrete objects, traditional flow-sensitive algorithms cannot kill the points-to information of field `f` of `o1` after the assignment statement at `L1` – it may unsoundly kill the points-to information of `r.f`. Hence the analysis concludes that `t1` may point to either `o2` or `o3` at the end of the program fragment (Figure 1(c)), although in any execution, `t1` can actually point to only `o3`. In general, `p` could have pointed to multiple abstract memory locations, which also would have made strong updates impossible for traditional flow-sensitive analyses.

In this paper we propose a different approach for flow-sensitive pointer analysis for Java programs that enables us to perform strong updates at indirect assignments effectively. Instead of a points-to graph, we compute a map from *access paths* to sets of abstract objects at each program statement. An access path is a local variable or a static field followed by zero or more field accesses. In the program fragment of Figure 1(a), the points-to set of the access path `p.f` can be strongly updated at `L1` regardless of whether `p` points to a single concrete memory location or not. On the other hand, the points-to set of `r.f` must be weakly updated at `L1` as `r.f` may alias to `p.f` at that program statement

[1] All analyses considered in this paper are *field-sensitive* [27].

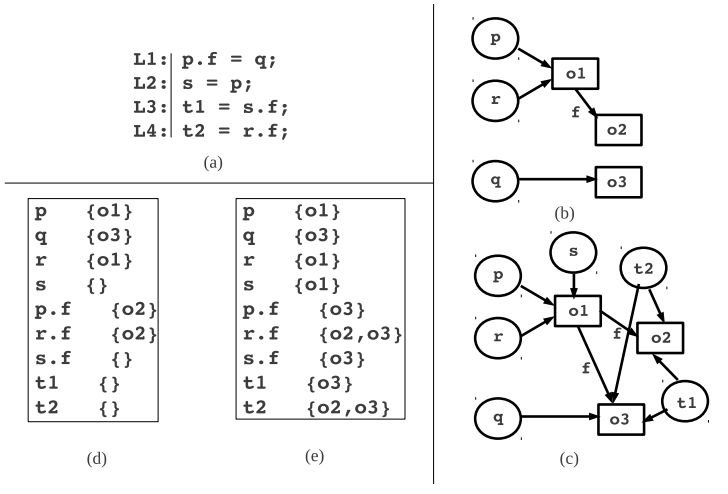


Fig. 1. (a) An example program fragment. (b) Points-to graph before the program fragment. (c) Points-to graph after the program fragment. (d) Our points-to information before the program fragment. (e) Our points-to information after the program fragment.

(two access paths may alias if they may refer to the same memory location). Note that we would strongly update `p.f` at `L1` even if `p` pointed to multiple abstract objects. We have observed that it is quite common to have (possibly interprocedural) program paths like Figure 1(a) which begin with an assignment to an access path and then subsequently read the access path, either directly or through an alias established by intervening pointer assignments. Our analysis targets such patterns and propagates the points-to sets from the initial assignment to the final read effectively through a series of strong updates.

While there has been earlier work on pointer analysis based on access paths for C programs [20,6], our approach differs from them in several ways. The key challenge to the scalability of such an analysis is the proliferation of access paths. In the presence of recursive data-structures, the number of access paths in a program can be infinite. As is standard, we bound the length of access paths using a user defined parameter l. The number of access paths however still grows exponentially with l, and it can be very expensive to maintain the full map of all access paths to their points-to sets at every program statement. One key feature of our algorithm is that we only need to store points-to sets of access paths that are *in scope* at a program statement. We also do further optimizations to reduce the size of the maps stored at each program statement as detailed in Section 3.3.

We bootstrap our analysis using a fast flow and context insensitive points-to analysis [1]. This base analysis is used in various stages of our analysis: to compute the set of access paths, to supply points-to sets of access paths longer than

the user-defined bound and to reach the fixpoint quickly by approximately precomputing the set of access paths modified at each program statement through aliasing.

We have implemented our analysis in the Chord framework [24]. The core of the analysis is written declaratively in Datalog [32]. Chord implements all Datalog relations using binary decision diagrams (BDD) [4] which helps in reducing the space required to store the points-to information. We have implemented our analysis both with and without context-sensitivity. Our analysis was run on eight moderately large Java programs with different values of l (the bound on the access path lengths) and we compared the precision of points-to sets and call-graphs with the traditional points-to graph based flow-insensitive [1] and flow-sensitive [16] analyses. On these benchmarks, for $l = 3$, our flow-sensitive and context-sensitive analysis shows a significant average improvement of 22% in precision over the flow-insensitive analysis with the same level of context-sensitivity, while terminating within reasonable time, whereas traditional flow-sensitive analysis has only less than 2% precision improvement over the flow-insensitive analysis and is much slower.

The rest of this paper is organized as follows. We give an overview of our technique with a couple of examples in Section 2. Section 3 describes our technique formally. We discuss an implementation of our technique and present empirical results in Section 4. Related works are discussed in Section 5. We discuss future directions and conclude with Section 6.

2 Overview

In this section, we informally describe the core of our algorithm using the program fragments of Figure 1(a) and Figure 2(a).

We first explain the intraprocedural part of our analysis using Figure 1. The intraprocedural analysis is an iterative dataflow analysis over the control flow graph (CFG). Our dataflow facts are maps from access paths to sets of abstract objects. We first compute a flow-insensitive points-to set for each variable. Let us assume that the points-to graph computed by the flow-insensitive analysis is as shown in Figure 1(c). The object `o1` has only one field, `f` and the objects `o2` and `o3` do not have any field. We also assume that the length of access paths is bound by the constant two. Hence the set of access paths in the program is {`p`, `q`, `r`, `s`, `p.f`, `r.f`, `s.f`, `t1`, `t2`}. Let us assume that the points-to information computed by our algorithm before `L1` is as shown in Figure 1(b). Figure 1(d) shows this information in our representation. The assignment at `L1` strongly updates the points-to set of `p.f` to {`o3`}. According to the flow-insensitive analysis, `r.f` may alias with `p.f` – hence the points-to set of `r.f` is updated to {`o2`,`o3`}. Although we could use our points-to information to detect the alias between `p.f` and `r.f` at `L1`, using a precomputed flow-insensitive analysis helps in reaching the fixpoint quickly. Note that this approximation may result in more weak updates, but does not affect strong updates. On the other hand, this approximation is

necessary for the scalability of our technique – without this approximation, six out of eight benchmarks did not terminate within 30 mins. Also note that `s.f` is not updated as it is not live at `L1`. The assignment at `L2` strongly updates the points-to sets of `s` and `s.f` with the points-to sets of `p` and `p.f` respectively. As in Java, a local variable like `s` cannot alias with any other access path, this assignment does not weakly update any access path. The assignments at `L3` and `L4` assigns the points-to sets of `s.f` and `r.f` to `t1` and `t2` respectively. The final points-to information is shown in Figure 1(e). The points-to set of `t1` in our analysis is more precise than the one computed by a traditional flow-sensitive analysis (Figure 1(c)).

We demonstrate the interprocedural analysis using the program fragment of Figure 2(a) (the statement at `LD` is commented out). In interprocedural analysis, at each program statement, we only store the information about access paths that are in scope at that statement. An access path is in scope at a statement if the access path starts with a variable local to the function containing the statement or with a static field. For the program fragment of Figure 2(a), the access paths of the outer function and the initial map at `LB` is shown in Figure 2(b). We use a mod/ref analysis based on the flow-insensitive points-to graph to determine that the points-to sets of `p.f` and `r.f` may be modified by the call to the function `setF`. On call to the function `setF`, `this`, `a`, and `this.f` are assigned the points-to sets of `p`, `q`, and `p.f` respectively. The assignment at `L8` strongly updates the points-to set of `this.f`. The map at `L9` is shown in Figure 2(c). On return to the outer function, as `p` and `this` must point to the same concrete object in all executions, the access path `p.f` can be strongly updated with the points-to set of `this.f`. On the other hand, as `r.f` may also be modified by the call to `setF` (because it is an alias of `p.f`) but `r` is not an actual parameter to the call, it is conservatively assigned its flow-insensitive points-to set. The map at `LC` is shown in Figure 2(d). The call to `getF` does not modify points-to sets of any access paths belonging to the outer function, but assigns the return variable of `getF` to `t1`. The final map is shown in Figure 2(e). Here also, the points-to set of `t1` is more precise than traditional flow-sensitive analyses which would not be able to do the strong update at `L8`.

If we include the statement at `LD`, the context-insensitive analysis would merge the points-to sets coming from `p.f` and `r.f` into the points-to set of `this.f` in function `getF`. This would make the points-to sets of both `t1` and `t2` to be {`o2`,`o3`}. Adding context-sensitivity would avoid this problem as two calls of `getF` would be distinguished by a context-sensitive analysis. For example, using a length 1 call-string as context would create two maps at `L3`, one tagged with call-site `LC` and mapping `this.f` to {`o3`} and another tagged with call-site `LD` and mapping `this.f` to {`o2`,`o3`}. On return, only the first map is used to assign points-to set of `t1`, making it {`o3`}. As Java programs use method calls extensively, we use call-string based context-sensitive analysis [30] to tag dataflow facts with fixed-length call-strings to distinguish between the calling contexts.

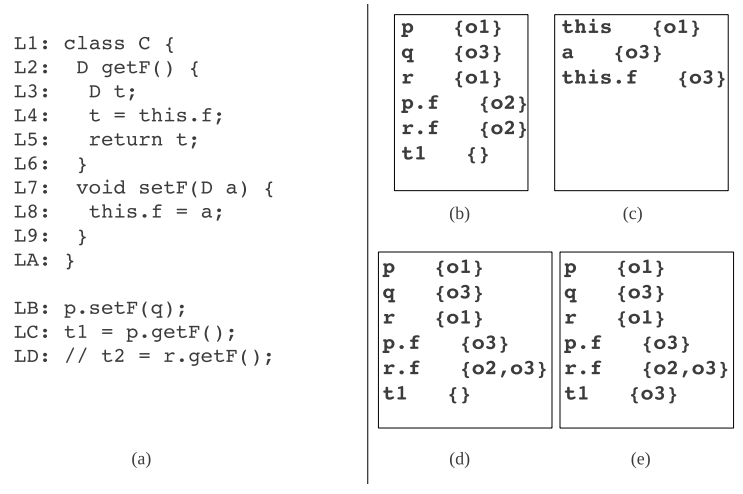


Fig. 2. (a) An example program fragment. (b) - (e) Points-to information at program statements `LB`, `L9`, `LC` and `LD`

3 Access Path-Based Flow-Sensitive Analysis

3.1 Background

Our input language is an intermediate code generated by the Chord framework [24] from Java bytecode. The intermediate language has at most one dereference per statement and is converted into partial *single static assignment* (SSA) form [11].

In SSA form, each variable is defined only once. If a variable is defined multiple times in the original program, each of those definitions is converted to a new version of that variable in SSA form. If multiple definitions of the same original variable reach a join point in the control flow graph, a ϕ statement is introduced at the join point. The ϕ statement merges the corresponding versions and creates a new version of the variable. In partial SSA form of Java programs, only the local variables are converted into SSA form. In a Java program, a local variable cannot be pointed by any variable or object field – hence there are no indirect assignments to these variables. Therefore, the definitions of such variables can be identified syntactically and can be converted to SSA form easily, without any prior pointer analysis.

Following are the different types of statements relevant to our pointer analysis and their representative forms. The variables `p`, `q`, `r` and the field `f` are of reference (pointer) type. `C` is the name of a class and `foo` is the name of a function. Without loss of generality, we consider function calls with only one parameter and ϕ statements merging two versions of a variable.

Allocation: `p = new C;`
Null assignment: `p = null;`
Copy: `p = q;`
Phi: `p = φ(q,r);`
Getfield: `p = q.f;`
Putfield: `p.f = q;`
Function call: `p = foo(q);`
Return: `return p;`

We do not consider static fields in this section. As array elements cannot be strongly updated in general, we treat array elements flow-insensitively. We assume that there is a `main` function designated as the entry point of the program. We also assume that all functions except the `main` function are virtual functions – the exact target function of a function call is determined by the type of the object pointed to by the first actual argument of the call. We also do not consider exceptions in this section. Our technique can easily be extended to handle all features of Java language and our implementation can take any Java program as input.

Like most other pointer analyses, we use allocation sites as abstract objects to represent all concrete objects allocated at that site. We represent the set of abstract objects in a program by *Objects*. In the rest of the paper, the word "object" means "abstract object", unless otherwise specified.

We bootstrap out analysis with a fast flow- and context-insensitive analysis [1]. This analysis computes the pointer information as a points-to graph G. We assume that the points-to graph supports two types of queries: $VarPts(G, v)$ returns the points-to set of the variable v and $ObjPts(G, o, f)$ returns the set of objects pointed to by the field f of the object o. Procedure 2 ($FIPts$) computes the set of objects pointed to by an access path in the points-to graph. The function $FIAlias$, defined below, detects if two different access paths may alias according to a points to graph G.

$$
\begin{aligned}
FIAlias_G = \lambda ap_1.\lambda ap_2. \quad & \text{if} \quad ap_1 \neq ap_2 \\
& \text{and} \quad ap_1 = ap_1'.f \text{ and } ap_2 = ap_2'.f \\
& \text{and} \quad FIPts(G, ap_1') \cap FIPts(G, ap_2') \neq \emptyset \\
& \qquad \text{then true else false.}
\end{aligned}
$$

3.2 Computing the Set of Access Paths

An *access path* is a local variable followed by zero or more field accesses[2]. More formally, given a program P with set of variables V and set of fields F, an access path is a member of the set $V(.F)^*$. The variable is called the *root* of the access path. The length of an access path is one more than the number of

[2] In general, an access path may start with a static field, we we do not consider static fields in this section for sake of simplicity.

Algorithm 1. Algorithm for computing set of access paths in a program

Input: Set of variables V, Points-to graph G, Map from abstract objects to actual types T, Bound on the lengths of access paths l.
Output: AP: Set of access paths in the function.

```
i = 1
AP ← V
newaps ← V
while i < k do
  nextaps ← ∅
  for all ap ∈ newaps do
    objset ← FIPts(G, ap)
  end for
  for all obj ∈ objset do
    objtype ← T(obj)
    for all field f of objtype do
      add ap.f to AP
      add ap.f to nextaps
    end for
  end for
  i ← i + 1
  newaps ← nextaps
end while
```

field accesses. In the presence of recursive data-structures, the set of access paths in a program can be infinite. We only consider access paths whose length is bound by a user-defined parameter l.

In order to compute the set of access paths in a program, we need to know that given an access path, which field accesses can be appended to it to generate longer access paths. It might be tempting to use the declared types of variables and fields to determine which field accesses are possible, but in the presence of inheritance, this approach runs into the following problem. Suppose `p` is a variable with declared type `A` and `q` is a variable with declared type `B`. Suppose `B` is a subtype of `A` and the class `B` has a field `f` which is not present in `A`. Suppose further that there is an assignment `p = q`. As there is no access `p.f`, we lose the points-to information of field `f` of the object pointed to by `p` after the assignment. Again, if there is downcast `q = (B)p`, we cannot determine the points-to information of `q.f` after the assignment. To avoid this problem, we use Algorithm 1 to compute the set of access paths in a program. This algorithm uses a flow and context insensitive points-to analysis to compute a points-to graph for the entire program. Given an access path, we traverse the points-to graph to determine the objects pointed to by the access path (Procedure 2). We extend the access path with all the fields of the actual types of these objects.

3.3 Intraprocedural Analysis

In this section, we assume that the input program has a single function with no call statements in order to focus on the intraprocedural analysis. Given a

Procedure 2. *FIPts*

Input: Points-to graph G, Access path ap.
Output: *objset*: Set of objects pointed to by ap in G.

```
if ap is a variable then
    objset ← VarPts(G, ap)
else if ap is of the form ap'.f then
    objset ← ⋃_{o∈FIPts(ap')} ObjPts(G, o, f)
end if
return objset
```

program function P, our intraprocedural analysis is an instance of iterative dataflow analysis [19] over the CFG $C = (N, E)$ of P, where N is the set of nodes of the CFG, representing the statements of the function and E is the set of control flow edges. We denote the root node of the CFG by n_0 and predecessor of a node n by $pred(n)$. The dataflow analysis $\mathcal{D} = (\mathcal{L}, \mathcal{F})$ of the function consists of a lattice $\mathcal{L}$ and a set of transfer functions $\mathcal{F}$, defined below.

Dataflow Lattice: The dataflow lattice $\mathcal{L} = (\mathcal{M}, \preceq)$ consists of a set of dataflow facts $\mathcal{M}$ and an ordering relation $\preceq$. The set $\mathcal{M}$ is the set of all maps from access paths to sets of abstract objects. Given two maps $m_1, m_2 \in \mathcal{M}$, $m_1 \preceq m_2$ iff for all access paths ap in AP, $m_1(ap) \subseteq m_2(ap)$. Naturally, the induced join operation is the point-wise union of points-to sets of all access paths in AP. Formally,

$$m_1 \sqcup m_2 = \lambda ap.\ (m_1(ap) \cup m_2(ap)).$$

Similarly, the greatest element of $\mathcal{M}$ is defined as $\top = \lambda ap.Objects$ and the least element as $\bot = \lambda ap.\emptyset$. As the sets AP and $Objects$ are both finite, the set $\mathcal{M}$ is also finite.

Transfer Functions: The transfer function for a CFG node describes how the statement at that node modifies a dataflow fact. Given a node n and an input map m_{in}, the output map m_{out} might map some access paths to different points-to sets than m_{in}. Table 1 describes, for each type of statement mentioned in Section 3.1 except for call and return statements, which access paths are mapped differently in m_{out} compared to m_{in}. For all other access paths ap, $m_{out}(ap) = m_{in}(ap)$. For all statements not listed in Table 1, $m_{out} = m_{in}$. In the table, a is an arbitrary non-empty field access sequence of the form $F(.F)^*$ where F is the set of fields in the program.

If a variable `p` is assigned a new object `o`, the points-to set of `p` contains only `o` and other access paths with `p` as root do not point to any object after the assignment. If the lhs of an assignment is a variable (say `p`) and the rhs is an access path (say ap), points-to sets of `p` and all access paths of the form `p.`a (a is any non-empty field sequence) are strongly updated with the points-to sets of ap and $ap.a$, respectively. For Getfield statements, as the length of access path

Table 1. Intraprocedural transfer functions. Column 1 lists types of statements. Column 2 lists the access path ap for which $m_{out}(ap)$ is different from $m_{in}(ap)$. Column 3 defines the points-to sets of m_{out} for such access paths. Here a is a non-empty field access sequence of the form $F(.F)^*$.

Statement	ap	$m_{out}(ap)$
`//abstract object o` `p = new C`	`p`	{`o`}
	`p.`a	$\emptyset$
`p = null`	`p`	$\emptyset$
	`p.`a	$\emptyset$
`p = q`	`p`	m_{in}(`q`)
	`p.`a	m_{in}(`q.`a)
`p =` ϕ`(q, r)`	`p`	m_{in}(`q`) $\cup$ m_{in}(`r`)
	`p.`a	m_{in}(`q.`a) $\cup$ m_{in}(`r.`a)
`p = q.f`	`p`	m_{in}(`q.f`)
	`p.`a	if `q.f.`$a \in AP$ then m_{in}(`q.f.`a) else $FIPts(G,$ `q.f.`$a)$
`p.f = q`	`p.f`	m_{in}(`q`)
	`p.f.`a	m_{in}(`q.`a)
	ap'`.f`	if $FIPts(G, ap') \cap FIPts(G,$ `p`$) \neq \emptyset$ then m_{in}(`q`) $\cup$ $m_{in}(ap'$`.f`$)$ else $m_{in}(ap'$`.f`$)$
	ap'`.f.`a	if $FIPts(G, ap') \cap FIPts(G,$ `p`$) \neq \emptyset$ then m_{in}(`q.`a) $\cup$ $m_{in}(ap'$`.f.`$a)$ else $m_{in}(ap'$`.f.`$a)$

on the lhs (say `p`) is shorter than the length of the access path on the rhs (say `q.f`), there might exist some access path of the form `p.`a such that there is no access path `q.f.`a, as its length might be more than the user-specified bound. In such cases, we use the flow-insensitive analysis to supply the points-to set for `p.`a. For all these statements, only access paths with `p` as root need to be updated. On the other hand, for Putfield statements, the lhs and its extensions are strongly updated, whereas the aliases of the lhs and their extensions are weakly updated. Note that, instead of our own analysis, we use a precomputed flow and context insensitive pointer analysis to detect these aliases. Using our analysis to detect these aliases would cause our analysis to find new access path assignments during the fixpoint computation. On the other hand, using a base pointer analysis enables us to precompute the set of direct and indirect access path assignments at all program statements. This approximation speeds up the fixpoint computation significantly. Note that this approximation may cause our analysis to perform more weak updates, but strong updates are not affected.

Multiple field accesses: Although the intermediate language described in Section 3.1 has at most one field access per statement, the original Java program

may have multiple field accesses per statement. For example, the statement `p.f.g = q;` is converted to the following sequence of statements: `t1 = p.f; t1.g = q;`, where `t1` is a temporary variable, defined only once. After the assignment to `t1.g`, according to the rules in Table 1, the access path `p.f.g` should be weakly updated, although according to the original program, it could have been strongly updated. As `t1` is defined only once in the partial SSA form and it is used immediately after the definition, we know that `t1` must point to the same object as `p.f` before the second assignment in the intermediate code. Hence we strongly update `t1.g` as well as `p.f.g` at the second statement.

Dataflow Equations: The dataflow fact at n_0 is $\perp$ and the transfer function for node n is denoted by F_n. We compute a dataflow fact at each node of the CFG. More specifically, we compute the least solution for X for the following set of dataflow equations:

$$\begin{aligned} X[n_0] &= \perp \\ \forall n \in (N - \{n_0\}) : X[n] &= \bigsqcup_{n' \in pred(n)} F_{n'}(X[n']) \end{aligned} \tag{1}$$

Optimizations: Although the set AP is finite, it can be very large – growing exponentially with the length of the access paths. This large size of AP in turn increases the space consumed by the elements of the set $\mathcal{M}$ as well as increases the time to compute the transfer functions. In order to reduce the sizes of dataflow facts, we perform two optimizations, described below.

In partial SSA form, each local variable is defined only once. Hence, in our analysis, the points-to set of a local variable can be changed by only one statement – the one defining it. Hence, instead of maintaining the map from local variables to its points-to set at every program point, we maintain a single map from local variables to their points-to sets in each function. The dataflow facts at each program point are maps from access paths with length greater than one to their points-to sets. This technique is adopted from [11].

We also observe that the points-to information of an access path is typically useful only at the program statements where it is *live*. A variable is live at a program statement if its definition reaches that statement and the variable is used subsequently in the function. An access path is live at a program statement if the root variable is live at that statement. At a program statement, we only maintain the map from live access paths to their points-to sets. As there is a global map for variables, the points-to information of variables are sound at all program statements, irrespective of whether the variable is live at that statement or not.

3.4 Interprocedural Analysis

The interprocedural analysis is an iterative dataflow analysis over the interprocedural control flow graph (ICFG), constructed by taking disjoint union of

individual control flow graphs of all functions and then adding call and return edges. A *call edge* connects a call statement to the first statement of the called function and a *return edge* connects a return statement to the statements following the corresponding call statements. Call and return edges for virtual functions are added *on-the-fly*; at a call site, if the receiver variable (the first actual parameter in our intermediate language) points to an abstract object `o`, the actual function to be called is determined by the actual type of `o`. As new objects are added to the points-to set of the receiver variable, new call and return edges are added – fixpoint is reached when no new objects are added to any points-to set *and* no new call/return edges are added to the ICFG. For the sake of simplicity, in this section we assume that the call and return edges are added a priori.

Context-Insensitive Analysis: We first consider context-insensitive analysis where dataflow facts are not distinguished by the calling contexts. The dataflow facts are the same as the intraprocedural analysis – maps from access paths to sets of abstract objects – but instead of maintaining the map for all access paths at all program statements, we only maintain the map for access paths that are in scope at that statement. An access path ap is *in scope* at a statement s if the root of the access path is a variable local to the function containing s[3]. The statement s may modify the points-to set of an access path ap' not in scope at s. As our analysis does not store the points-to information of ap' at s, the change in the points-to set of ap' is not reflected immediately after s – it is updated on return to the function where ap' is in scope. The ability to discard the points-to information of access paths at program statements where they are not in scope but still soundly update them on return is the key to the scalability of our analysis.

Call statements in the ICFG have multiple outgoing edges – one edge to the next statement of the same function and (potentially multiple) call edges. Similarly, the return statements may also have multiple outgoing edges – one return edge for each calling method. The transfer functions for call and return statements propagate different dataflow information along different edges. Note that the first statement of a function acts as a join node in the ICFG, joining the call edges from different call-sites. In an ICFG, the statement following a call statement in the same function also joins the return edges from the called functions with the CFG edge. The transfer functions for all statements described in Section 3.3 remain unchanged, but to maintain uniformity with the call and return statements, we add an outgoing edge as a second parameter to the transfer function. For all statements n except for call and return, $F_n(m, e)$ is same as $F_n(m)$ as defined in Section 3.3, where e is an outgoing edge of node n and m is a dataflow fact.

Table 2 defines the transfer function for a call statement `foo(q)` along the CFG edge and along a call edge to the function `foo` with formal parameter `p` (as before, a denotes an arbitrary non-empty field sequence). Along the call edge,

[3] An access path with a static field as root is in scope everywhere, but we do not consider static fields in this section.

it initializes the points-to sets of access paths that have formal parameters of the called function as roots. Along the CFG edge, it kills the points-to sets of access paths that may be updated inside the called function. We use a mod/ref analysis based on the flow-insensitive points-to graph to determine if an access path may be modified by a called function. A function `foo` *may modify* a field `f` of an abstract object `o` if one of the following is true:

1. There is a statement in `foo` writing to `v.f` such that $\texttt{o} \in \mathit{FIPts}(G, v)$ (G is the flow-insensitive points-to graph).
2. `foo` calls a function which may modify the field `f` of the object `o`.

An access path l`.f` may be modified by a call to `foo` if $o \in \mathit{FIPts}(G, l)$ and `foo` may modify field `f` of object `o`. If an access path ap may be modified by a call to `foo`, we write $\mathit{MayMod}(\texttt{foo}, ap)$. Note that the output map along CFG edge only contains access paths local to the calling function, whereas the output map along the call edge only has access paths local to the called function.

Table 2. Transfer function for call statement `foo(q)` along the call edge to function `foo` with formal parameter `p` and along the CFG edge. Here a is an arbitrary non-empty field sequence.

Edge	ap	$m_{out}(ap)$
Call edge	`p`	$m_{in}(\texttt{q})$
	`p.`a	$m_{in}(\texttt{q}.a)$
	other	$\emptyset$
CFG edge	ap s.t. $\mathit{MayMod}(\texttt{foo}, ap)$	$\emptyset$
	other	$m_{in}(ap)$

Table 3 describes the transfer function for the return statement `return r` along the return edge corresponding to the call statement `s = foo(q)`. The formal parameter of the function `foo` containing the return statement is `p`. The output map only contains access paths local to the calling function. The return statement updates the points-to sets of the access paths rooted at `s` with the points-to sets of the corresponding access paths rooted at `r`.

We use the transfer function of the return statement to soundly update the points-to sets of access paths of the calling function that could have been modified during the execution of `foo`. As the input language is in partial SSA form, the formal parameter of a function cannot be reassigned inside the function. Hence, in every execution, the formal parameter `p` and the actual parameter `q` must point to the same concrete object throughout the execution of `foo`. Therefore, on return, the value of `q.`a would be same as the value of `p.`a at the return statement. Hence, if `q.`a may be modified by `foo`, the transfer function of the return statement assigns the points-to sets of access paths rooted at `p` to the points-to sets of the corresponding access paths rooted at `q`. As the points-to sets of access paths that may be modified by the called function are killed at the

call statement (Table 2), this results in strong updates of such access paths after the join of the return edge with the CFG edge of the calling function. Any access paths that can be modified by `foo` but not rooted at the actual parameter of the call are assigned their flow-insensitive points-to set after the call statement. Any access paths that cannot be modified by `foo` are assigned empty sets. As the points-to sets of such access paths are not killed by the call statements, they retain their points-to set prior to the call to `foo` after the join with the CFG edge.

Table 3. Transfer function for return statement `return r` along return edge corresponding to the call statement `s = foo(q)`. The formal parameter of the function `foo` containing the return statement is `p`. Here G is a points-to graph computed by the base analysis and a is an arbitrary non-empty field sequence.

ap	$m_{out}(ap)$
`s`	$m_{in}(\texttt{r})$
`s`.a	$m_{in}(\texttt{r}.a)$
`q`.a	if $MayMod(\texttt{foo}, ap)$ then $m_{in}(\texttt{p}.a)$ else $\emptyset$
other	if $MayMod(\texttt{foo}, ap)$ then $FIPts(G, ap)$ else $\emptyset$

The transfer function for return statement can be imprecise for many access paths; but it can perform strong updates for access paths rooted at the actual parameters. In Java programs, often such access paths are read later, either directly or through an alias established though intervening pointer assignments. Our technique can have the benefit of strong updates in such cases.

The interprocedural dataflow analysis computes the least solution of the following set of dataflow equations. Here n_0 is the root node of the ICFG, i.e. the root node of the `main` function and $\langle n', n\rangle$ denotes the ICFG edge between nodes n' and n.

$$\begin{aligned} X[n_0] &= \bot \\ \forall n \in (N - \{n_0\}) : X[n] &= \bigsqcup_{n' \in pred(n)} F_{n'}(X[n'], \langle n', n\rangle) \end{aligned} \tag{2}$$

Context-Sensitive Analysis: For context-sensitivity, we use the standard call-string approach [30] with finite sequence of call-sites as contexts.

We first describe the context-sensitive technique with unbounded call-strings. Let $\mathcal{D} = (\mathcal{L}, \mathcal{F})$ be the underlying dataflow analysis with $\mathcal{L} = (\mathcal{M}, \preceq)$. Let $C^* = (N, E)$ denote the ICFG of the program. We define a *call-string* γ as a

(possibly empty) sequence of call statements. Let Γ be the set of all such call-strings. The empty call-string is denoted by ϵ. The length of a call-string γ is denoted by $|\gamma|$. The ith component of γ is denoted by $\gamma[i]$ and the substring from ith to jth component (both inclusive) is denoted by $\gamma[i..j]$. The operator "." denotes the string append operation.

The call-string approach defines a new dataflow analysis framework $\mathcal{D}^* = (\mathcal{L}^*, \mathcal{F}^*)$, where $\mathcal{L}^* = (\mathcal{M}^*, \preceq^*)$. The domain $\mathcal{M}^*$ is the space of all maps from Γ into $\mathcal{M}$. The ordering in $\mathcal{L}^*$ is the point-wise ordering on $\mathcal{L}$, i.e. for $\xi_1, \xi_2 \in \mathcal{M}^*, \xi_1 \preceq^* \xi_2$ iff $\forall \gamma \in \Gamma, \xi_1(\gamma) \preceq \xi_2(\gamma)$.

In order to define the flow functions, we first define a partial binary operator $\circ : \Gamma \times E \to \Gamma$ in the following way:

$$\gamma \circ \langle n, n' \rangle = \begin{cases} \gamma \cdot n & \text{if } \langle n, n' \rangle \text{ is a call edge} \\ \gamma[1..|\gamma| - 1] & \text{if } \langle n, n' \rangle \text{ is a return edge and } \gamma[|\gamma|] \text{ is the corresponding call statement} \\ \gamma & \text{otherwise} \end{cases}$$

A flow function $F_n^* \in \mathcal{F}^*$, where $n \in N$, is a function from $\mathcal{M}^* \times E$ to $\mathcal{M}^*$, defined below:

$$F_n^*(\xi, e)(\gamma) = \begin{cases} F_n(\xi(\gamma'), n) & \text{if there exists a unique } \gamma' \text{ such that } \gamma = \gamma' \circ e \\ \bot & \text{otherwise} \end{cases}$$

The solution of the analysis is the least solution of the dataflow equations corresponding to the lattice and transfer functions defined above.

As the set of unbounded call-strings is infinite, we use a k length suffix of the unbounded call-string as approximate call-string. Details of the call-string approach can be found in [30].

4 Implementation and Experimental Results

We have implemented our analysis within the Chord framework [24]. Chord encodes program structures such as CFG, assignment statements and type hierarchies as relations and implements them using BDDs [4]. We use Chord's built-in flow and context insensitive pointer analysis as our base analysis. Our frontend, written in Java, computes the set of access paths and other relevant program information as relations implemented using BDDs. The core analysis is written declaratively in Datalog [32] which takes the relations produced by the frontend as input. Our implementation first converts each assignment of the program into multiple assignments to access paths, capturing all possible strong and weak updates of access paths by that assignment. We use the precomputed base pointer analysis to perform the possible weak updates. The next phase of the analysis computes the flow-sensitive points-to sets for access paths and constructs the call-graph on-the-fly. Chord uses `bddbddb` [36] for fixpoint computation of the Datalog analyses. We have implemented our analysis both with and without context-sensitivity. The context-sensitive analysis is a call-string analysis with call-string length 1.

Table 4. Characteristics of the benchmarks

Benchmarks	Int. Code Stmts	Classes	Methods	AP($l = 2$)	AP($l = 3$)	Contexts
polyglot	123789	1474	5933	14651	113354	24690
jlex	126661	1411	5708	21149	137578	10037
javacup	117804	1389	5498	22018	136773	11721
jtopas	120499	1432	5736	17308	137714	9068
jdom	119437	1443	5610	13704	107536	8473
jasmin	123490	1381	5174	39480	179780	11307
jjdoc	85256	1270	3701	31001	63540	9485
jjtree	93958	1376	4219	32343	58018	11155

We have run our analysis on eight moderately large Java benchmarks. We have used a laptop with 2.3 GHz Core i5 processor with 3GB memory for our experiments. We have used OpenJDK 1.6 as our JDK. Table 4 shows the sizes of intermediate code, number of classes and methods, number of access paths with bound 2 and 3 and the number of contexts for these benchmarks. As our analysis includes the Java libraries as well, we report the number of lines in the intermediate code within the scope of the analysis as constructed by Rapid Type Analysis, instead of the size of source code of the application program only. Note that the number of access paths grows rapidly with respect to l.

We compare the precision of our analysis with that of the points-to graph based flow-insensitive [1] and flow-sensitive [16] analyses. As measurement of precision, we use the sizes of the points-to sets of the local variables. Note that due to the partial SSA form, our flow-sensitive analysis stores a single points-to set for each local variable for the entire program (cf. Section 3.3). Hence, we can directly compare the total size of the points-to sets of local variables obtained by our analysis with that of the points-to graph based flow-insensitive and flow-sensitive analyses. Note that the heap-based memory locations are represented differently in our analysis than points-to graph based analyses; hence we do not compare the sizes of such memory locations directly. Nevertheless, as our intermediate code is single-dereference based, contents of any heap location must be copied to a local variable before it can be dereferenced. Thus any change in the sizes of the points-to sets of heap locations are reflected in the sizes of the points-to sets of the local variables.

We also construct call-graphs of the input programs using the pointer information. Nodes of a call-graph are methods of the input program. If a method `m` calls a method `n`, the call-graph has an edge from `m` to `n`. For virtual calls, the actual method to be called at run-time depends on the object pointed to by the first actual parameter of the call site – hence a precise pointer analysis may reduce the number of edges of the call-graph (henceforth we refer to the number of edges of a call-graph as its *size*).

We observe that the precision improvement of our analysis over traditional flow-insensitive analysis without context-sensitivity is very small – only 5% on average for points-to sets of local variables and 6% on average for call-graphs.

This is expected for Java programs as they use method calls extensively to access fields and without context-sensitivity, the points-to sets of access paths in those methods are merged for all calls. Such a situation is shown in Section 2.

With a call-string length of 1, our flow- and context-sensitive analysis shows significant precision improvement over the flow-insensitive analysis with the same level of context-sensitivity. The flow-insensitive analysis also uses the partial SSA form, hence it already has the benefit of flow-sensitivity for local variables [11] – our analysis shows precision improvements on top of that. On the other hand, the points-to graph based flow-sensitive analysis shows only less than 2% improvement over the flow-insensitive analysis. Hence, the precision improvement of our analysis comes only from strong updates of heap-residing pointers. Figure 3 shows the precision improvements for points-to sets of local variables over the flow-insensitive analysis on eight benchmarks for different bounds on the lengths of the access paths. It also shows the result for the traditional points-to graph based flow-sensitive analysis. On the average, our flow-sensitive analysis reduces the sizes of points-to sets of local variables by 22% with $l = 3$ and by 16% with $l = 2$ over the flow-insensitive analysis with partial SSA form. All analyses are context-sensitive with call-string length 1.

We also report the reduction in call-graph size over the flow-insensitive analysis in Figure 4. The average reduction in call-graph size is 30% for $l = 3$ and 26% for $l = 2$. Table 5 shows the time taken by the flow-insensitive analysis, the points-to graph based flow-sensitive analysis and our flow-sensitive analysis (for $l = 2$ and $l = 3$) on these benchmarks. On the other hand, the traditional flow-sensitive analysis does not show any non-trivial reduction in the call-graph sizes with respect to the flow-insensitive analysis – hence we omit the comparison with such analysis in Figure 4 for the sake of clarity.

On the average, our analysis has a slowdown of 9.2X with $l = 3$ and of 5.8X with $l = 2$ with respect to the flow-insensitive analysis, but it is much faster than the points-to graph based flow-sensitive analysis.

Table 5. Time taken by flow-insensitive analysis (FI time), points-to graph based flow-sensitive analysis and our our flow-sensitive analysis (with access path length 2 and 3). All analyses are context-sensitive with call-string length 1.

Benchmarks	FI time	FS time (points-to graph)	FS time ($l = 2$)	FS time ($l = 3$)
polyglot	52	756	201	411
jlex	51	858	220	421
javacup	61	838	477	808
jtopas	47	820	306	390
jdom	49	609	297	378
jasmin	58	824	388	552
jjdoc	36	589	166	286
jjtree	44	610	228	424
Average	**49.8**	**738.0**	**292.9**	**458.8**

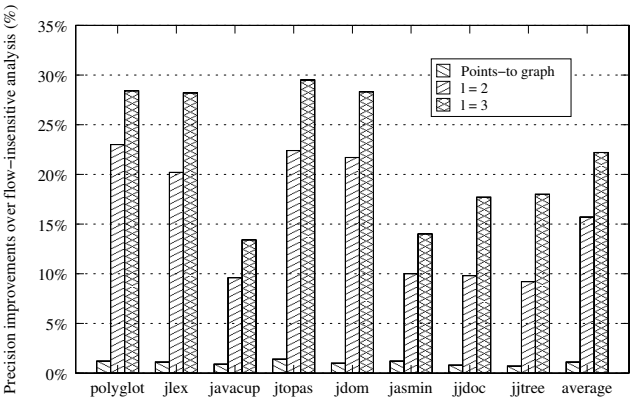


Fig. 3. Reduction in sizes of points-to sets by points-to graph based flow-sensitive analysis and our flow-sensitive analysis (with access path lengths 2 and 3) over flow-insensitive analysis with partial SSA form. All analyses are context-sensitive with call-string length 1.

These empirical results show that our analysis has significant precision improvement over the flow-insensitive analysis due to the strong updates of heap-based pointers which can not be achieved by traditional flow-sensitive analysis. The time taken by our analysis is also reasonable compared to the traditional flow-sensitive analysis.

5 Related Work

Flow-Sensitive Pointer Analysis: Pointer analysis is a fundamental static analysis with a long history. Early flow-sensitive pointer analyses [20,6] explicitly stored the pairs of access paths that might alias with each other. These works do not focus on dynamically allocated data-structures. The experimental results are preliminary. Emami et al. [7] presents a flow-sensitive and context-sensitive pointer analysis that uses points-to information between abstract stack locations as dataflow facts. They mark each points-to relation as *may* or *must* to perform strong updates on indirect assignments. Some analyses [37,35] use an intraprocedural flow-sensitive analysis to build procedure summaries that are instantiated at call-sites, but none of these analyses can perform strong updates on pointers residing in the heap. Hasti and Horwitz [13] incrementally build an SSA representation from the aliases already discovered – a flow-insensitive analysis on the SSA form gives the same benefit as a flow-sensitive one for the memory location already converted into SSA form. It remains an open question whether the fixpoint of this technique matches the result of a flow-sensitive analysis. Hind et al. [15] express the flow-sensitive pointer analysis as an iterative

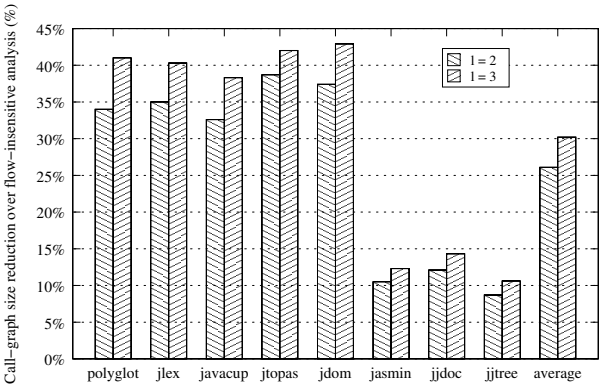


Fig. 4. Reduction in call-graph sizes by our flow-sensitive analysis (with access path lengths 2 and 3) over flow-insensitive analysis with partial SSA form. All analyses are context-sensitive with call-string length 1.

dataflow analysis over the CFG. They use a *compact representation*, essentially a form of points-to graph, as dataflow facts. In order to perform strong updates at indirect assignments, they keep track of whether a pointer points to a single concrete object.

More recently, researchers have focused on improving the scalability of flow-sensitive pointer analysis for C programs. Hardekopf and Lin [11] proposed a semi-sparse analysis which uses a partial SSA form for top-level variables and a sparse evaluation graph to eliminate irrelevant nodes. This approach is further extended in [12], where a flow-insensitive analysis is used to compute approximate def-use pairs, which helps in speeding up the sparse analysis in a later stage. The technique proposed by Yu et al. [38] first partitions the pointers into different levels by a flow-insensitive analysis such that there is an unidirectional flow of value from higher to lower level. Once the higher level variables are analyzed, the result can be used to build SSA representation for the lower level variables. Lhotak et al. [21] performs flow-sensitive analysis only on those memory locations which can be strongly updated. Li et al. [22] reduce the flow-sensitive pointer analysis problem to a graph reachability problem in a value flow graph which represents dependence between pointer variables. All these analyses do not perform strong updates for heap-residing pointers.

Zhu [39] uses BDDs to improve scalability of flow and context sensitive pointer analysis. This technique cannot perform any strong updates as querying whether a variable points to a single object is not efficiently supported by BDDs. As our technique does not need such uniqueness queries to perform strong updates, we can use BDDs efficiently.

Fink et al. [8] proposes a flow and context sensitive analysis for typestate verification. They use access paths to determine if a concrete object is *live*, i.e.

accessible via some access paths. They use a uniqueness analysis to identify abstract objects that represents a single live concrete object so that strong updates can be applied to those objects. It is not known how many strong updates can be done for general pointer analysis using their technique. Our analysis does not rely on the uniqueness of an abstract object to perform strong updates.

Bootstrapping: Several pointer analysis techniques use a fast and imprecise analysis to bootstrap their own analysis. Kahlon [18] uses a fast and imprecise analysis to partition the code such that each part can be analyzed independently. Similarly, Fink et al. [8] apply successively more precise techniques to smaller parts of the code. As mentioned before, Yu et al. [38] uses a flow-insensitive analysis to partition the pointers into different levels. Similarly, Hardekopf et al. [12] uses an auxiliary flow and context insensitive analysis to compute the approximate def-use chains.

Declarative Pointer Analysis: Whaley [33] developed `bddbddb`, a framework for implementing program analyses declaratively in Datalog [32] and implemented a context-sensitive pointer analysis using this framework. Later, Bravenboer and Smaragdakis [3] implemented several context-sensitive analyses in Datalog. All these algorithms are flow-insensitive.

6 Conclusion and Future Work

In this paper, we have presented a flow-sensitive pointer analysis algorithm for Java that can perform strong updates on pointers residing in the heap. Our implementation scales for moderately large benchmarks. Our flow and context sensitive analysis shows significant precision improvement over the flow-insensitive analysis with partial SSA form as well as traditional points-to graph based flow-sensitive analysis, with same level of context-sensitivity, on those benchmarks.

In future, we would like to improve the scalability of our analysis further by implementing it over sparse evaluation graphs [11]. We would also like to incorporate different types of context-sensitivity in our analysis such as object-sensitivity [23].

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
2. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using bdds. In: PLDI 2003: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 103–114. ACM, New York (2003)
3. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 243–262. ACM, New York (2009)

4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. 35, 677–691 (1986)
5. Chang, W., Streiff, B., Lin, C.: Efficient and extensible security enforcement using dynamic data flow analysis. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 39–50. ACM, New York (2008)
6. Choi, J.-D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1993, pp. 232–245. ACM, New York (1993)
7. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pp. 242–256. ACM, New York (1994)
8. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. 17, 9:1–9:34 (2008)
9. Guyer, S.Z., Lin, C.: Error checking with client-driven pointer analysis. Sci. Comput. Program. 58, 83–114 (2005)
10. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. SIGPLAN Not. 42(6), 290–299 (2007)
11. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 226–238. ACM, New York (2009)
12. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 289–298 (April 2011)
13. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: PLDI 1998: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 97–105. ACM, New York (1998)
14. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 254–263. ACM, New York (2001)
15. Hind, M., Burke, M., Carini, P., Choi, J.-D.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21, 848–894 (1999)
16. Hind, M., Pioli, A.: Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 57–81. Springer, Heidelberg (1998)
17. Hind, M., Pioli, A.: Which pointer analysis should i use? In: ISSTA 2000: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 113–123. ACM, New York (2000)
18. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: PLDI 2008: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 249–259. ACM, New York (2008)

19. Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 194–206. ACM, New York (1973)
20. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. In: PLDI 1992: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, pp. 235–248. ACM, New York (1992)
21. Lhoták, O., Chung, K.-C.A.: Points-to analysis with efficient strong updates. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 3–16. ACM, New York (2011)
22. Li, L., Cifuentes, C., Keynes, N.: Boosting the performance of flow-sensitive points-to analysis using value flow. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, SIGSOFT/FSE 2011, pp. 343–353. ACM, New York (2011)
23. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. 14, 1–41 (2005)
24. Naik, M.: jchord: A static and dynamic program analysis platform for java, http://code.google.com/p/jchord/
25. Nystrom, E.M., Kim, H.-S., Hwu, W.-m.W.: Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 165–180. Springer, Heidelberg (2004)
26. Pearce, D.J.: Some directed graph algorithms and their application to pointer analysis. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing (2005)
27. Pearce, D.J., Kelly, P.H., Hankin, C.: Efficient field-sensitive pointer analysis of c. ACM Trans. Program. Lang. Syst. 30(1) (November 2007)
28. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP 2001: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 12–23. ACM, New York (2001)
29. Shapiro II, M., Horwitz, S.: The Effects of the Precision of Pointer Analysis. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 16–34. Springer, Heidelberg (1997)
30. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis, ch.7, pp. 189–234. Prentice-Hall, Englewood Cliffs (1981)
31. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL 1996: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32–41. ACM, New York (1996)
32. Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies. W. H. Freeman & Co., New York (1990)
33. Whaley, J.: Context-Sensitive Pointer Analysis using Binary Decision Diagrams. PhD thesis, Stanford University (March 2007)
34. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI 2004, pp. 131–144. ACM, New York (2004)
35. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. In: OOPSLA 1999: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 187–206. ACM, New York (1999)
36. Whaley, J., Unkel, C., Lam, M.S.: A bdd-based deductive database for program analysis (2004), http://suif.stanford.edu/bddbddb

37. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for c programs. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI 1995, pp. 1–12. ACM, New York (1995)
38. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2010, pp. 218–229. ACM, New York (2010)
39. Zhu, J.: Towards scalable flow and context sensitive pointer analysis. In: Proceedings of the 42nd Annual Design Automation Conference, DAC 2005, pp. 831–836. ACM, New York (2005)

Application-Only Call Graph Construction

Karim Ali and Ondřej Lhoták

David R. Cheriton School of Computer Science, University of Waterloo

Abstract. Since call graphs are an essential starting point for all interprocedural analyses, many tools and frameworks have been developed to generate the call graph of a given program. The majority of these tools focus on generating the call graph of the whole program (i.e., both the application and the libraries that the application depends on). A popular compromise to the excessive cost of building a call graph for the whole program is to ignore all the effects of the library code and any calls the library makes back into the application. This results in potential unsoundness in the generated call graph and therefore in any analysis that uses it. In this paper, we present CGC, a tool that generates a sound call graph for the application part of a program without analyzing the code of the library.

1 Introduction

A call graph is a necessary prerequisite for most interprocedural analyses used in compilers, verification tools, and program understanding tools [19]. However, constructing a sound, precise call graph for even a small object-oriented program is difficult and expensive. For example, constructing the call graph of a Java "Hello, World!" program using SPARK [20] can take up to 30 seconds, and produces a call graph with 5,313 reachable methods and more than 23,000 edges. The key reason is dynamic dispatch: the target of a call depends on the runtime type of the receiver of the call. Because the receiver could have been created anywhere in the program, a sound algorithm must either analyze the whole program [1,6,17,21,34], or make very conservative assumptions about the receiver type (e.g., Class Hierarchy Analysis [9]). Additionally, due to the large sizes of common libraries, whole-program analysis of even trivial programs is expensive [7,27,28]. Practical programs generally have many library dependencies, and in many cases, the whole program may not even be available for static analysis. Our aim is to construct sound and precise call graphs for the application part of a program without analyzing the libraries that it depends on.[1]

Construction of partial call graphs is an often-requested feature in static analysis frameworks for Java. On the mailing list of the SOOT framework [34], which analyzes the whole program to construct a call graph, dozens of users have requested partial call graph construction [4]. One popular approach for generating

[1] In the rest of this paper, we will use the singular "library" to mean all of the libraries that a program depends on.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 688–712, 2012.

partial call graphs, used for example in the WALA framework [17], is to define an analysis scope of the classes to be analyzed. The analysis scope then represents the application part of the program. The effects of code outside this scope (i.e., the library) are ignored. As a consequence, the generated call graph lacks edges representing call-backs from the library to the application. Methods that should be reachable due to those call-back edges are ignored as well. Since this approach ignores the effects of the library code, any store or load operation in the library that involves an application object is ignored. Therefore, the points-to sets of the application objects will be incomplete, potentially causing even more call graph edges to be missing.

In contrast, we aim to produce a partial call graph that soundly overapproximates the set of targets of every call site in the analysis scope, and the set of reachable methods in the analysis scope. Our call graph uses a single summary node to represent all methods in the library. However, the analysis should be accurate for the application code. The goal of our work is to make less conservative assumptions about the library code, which is not analyzed, while still generating a precise and sound call graph for the application.

The essential observation behind our approach is that the division between an application and its library is not arbitrary. If the analysis scope could be any set of classes, then the call graph would necessarily be very imprecise. In particular, a sound analysis would have to assume that the unanalyzed code could call any non-private method and modify any non-private field in the analysis scope.[2]

A realistic yet very useful assumption is that the code of the library has been compiled without access to the code of the application. We refer to this as the *separate compilation assumption*. From this, we can deduce more specific restrictions on how the library can interact with the application, which we will explain in detail in Section 3. In particular, the library cannot call a method, access a field, or instantiate a class of the application if the library author does not know the name of the method, field, or class. It is theoretically possible to discover this information using reflection, and some special-purpose "libraries" such as JUnit [18] actually do so. We assume that such reflective poking into the internals of an application is rare in most general libraries.

In this paper, we evaluate the hypothesis that this assumption of separate compilation is sufficient to construct precise call graphs. We provide a prototype implementation for call graph construction, CGC, that uses a pointer analysis based on the separate compilation assumption. We evaluate soundness by comparing against the dynamic call graphs observed at run time by *J [11]. Since a dynamic call graph does not represent all possible paths of a program, it does not make sense to use it to evaluate the precision of a static call graph. Therefore, we evaluate precision by comparing against call graphs constructed by whole-program analysis (using both the SPARK [20] and DOOP [6] call graph construction systems). We also compare the performance of our prototype

[2] Some field modifications could theoretically be ruled out if an escape analysis determined that some objects are not reachable through the heap from any objects available to the unanalyzed code.

partial call graph construction system with whole-program call graph construction. However, our prototype implementation is optimized for adaptability and for producing call graphs comparable to those of other frameworks in terms of soundness and precision and not specifically for performance. Given the positive research results from our prototype, an obvious next implementation step would be to optimize and embed the analysis within popular analysis frameworks such as SPARK, DOOP, and WALA.

In summary, this paper makes the following contributions:

- It identifies the *separate compilation assumption* as key to partial call graph construction, and specifies the assumptions about the effects of library code that can be derived from it.
- It presents our prototype implementation of a partial call graph construction system, CGC.
- It empirically shows that the separate compilation assumption is sufficient for constructing precise and sound application-only call graphs.

2 Background

2.1 Call Graph Construction

The targets of method calls in object-oriented languages are determined through dynamic dispatch. Therefore, a precise call graph construction technique requires a combination of two inter-related analyses: it must determine the targets of calls, and also determine the run-time types of referenced objects (i.e., points-to analysis). In Figure 1, both analyses are divided further and their dependencies are made more explicit. Determining the targets of calls is divided into two relations: *reachable methods* and *call edges*. On the other hand, the points-to analysis is defined by two other relations: *points-to sets* and *points-to constraints*.

The main goal of a call graph construction algorithm is to derive the call edges relation. A call edge connects a call site, which is an instruction in some

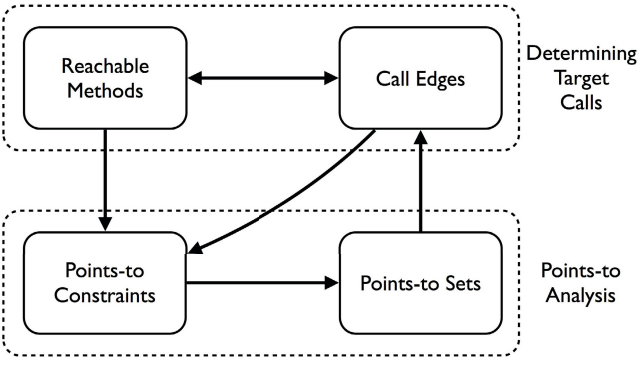


Fig. 1. Inter-dependent relations that make up a call graph construction algorithm

method, to a method that may be invoked from that call site. Figure 1 shows that the call edge relation depends on two relations: reachable methods and points-to sets. First, we are only interested in call sites that may actually be executed and are not dead code. A precise call graph construction algorithm therefore keeps track of the set of methods that are transitively reachable from the entry points of the program, e.g., its `main()` method. Second, the target of a given call depends on the run-time type of the receiver of the call. A precise call graph construction algorithm therefore computes the points-to sets abstracting the objects that each variable could point to. There are two common methods of abstraction to represent objects: either by their allocation site (from which their run-time type can be deduced) or by their run-time type. Thus, the points-to set of a variable at a call site indicates the run-time types of the receiver of that call.

Points-to sets are computed by finding the least fixed-point solution of a system of subset constraints that model all possible assignments between variables in the program. Thus, an abstract object "flows" from its allocation site into the points-to sets of all variables to which it could be assigned. Eventually, the abstract object reaches all of the call sites at which its methods could be called. The dependency of the points-to set relation on the call edges relation is illustrated in Figure 1. The calculation of the points-to sets is subject to the points-to constraints. The points-to constraints model intra-procedural assignments between variables due to explicit instructions within methods. They also model inter-procedural assignments due to parameter passing and returns from methods. Since only intra-procedural assignments in reachable methods are considered, the set of points-to constraints depends on the set of reachable methods. The set of call edges is another dependency because it determines the inter-procedural assignments.

Finally, the set of reachable methods depends, of course, on the set of call edges. A method is reachable if any call edge leads to it. A precise call graph construction algorithm computes these four inter-dependent relations concurrently until it reaches a mutual least fixed point. This is often called *on-the-fly* call graph construction.

2.2 Partial Call Graph Construction

If a sound call graph is to be constructed without analyzing the whole program, conservative assumptions must be made for all four of the inter-dependent relations in Figure 1. A sound analysis must assume that any unanalyzed methods could do "anything": they could arbitrarily call other methods and assign arbitrary values to fields. Due to the dependencies between the four relations, imprecision in any one relation can quickly pollute the others.

Our call graph construction algorithm computes precise information for the application part of the call graph, but uses summary nodes for information about the library. It assumes that all library methods are reachable, and uses a single summary "method" to represent them. Calls from application methods to library methods and vice versa are represented as call edges to or from the library

```
public class Main {
   public static void main(String[] args) {
       MyHashMap<String,String> myHashMap = new MyHashMap<String,String>();
       System.out.println(myHashMap);
   }
}

public class MyHashMap<K,V> extends HashMap<K,V> {
   public void clear() { }
   public int size() { return 0; }
   public String toString() { return "MyHashMap"; }
}
```

Fig. 2. A sample Java program that will be used for demonstration

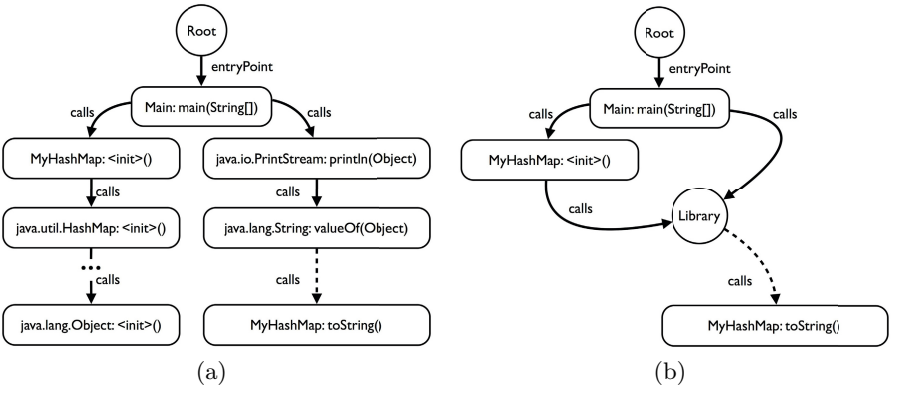


Fig. 3. Two branches from the call graph for the sample program in Figure 2 as generated by (a) SPARK and (b) CGC. The dashed line represents a call from a library method to an application method (i.e., a library call-back).

summary node. A call edge is created for each possible call between application methods, but no edges are created to represent calls within the library. It is implicitly assumed that any library method could call any other library method. Similarly, a single summary points-to set is used to represent the points-to sets of all variables within the library. Intra-library pointer flow, however, is not tracked precisely.

Figure 2 shows a sample Java program that we will use to demonstrate our analysis. Figure 3 compares two branches from the call graphs generated for this sample program by (a) SPARK and (b) CGC. We computed both branches by following the paths from the entry point method of the call graph, `Main.main()`, using the `CallGraphView` tool that comes with PROBE [19]. In Figure 3(a), we can see that the first branch shows all of the call edges from the method `MyHashMap.<init>()` all the way up to `java.lang.Object.<init>()`. Moreover,

the second branch in the call graph shows the call edges between library methods, e.g., the call edge from the method `java.io.PrintStream.println(Object)` to the method `java.lang.String.valueOf(Object)`. The target of that edge calls back to the application method `MyHashMap.toString()`.

On the other hand, Figure 3(b) shows how CGC represents the same branches. All of the edges beyond the predefined point of interest of the user (i.e., the application classes `MyHashMap` and `Main`) are considered as part of the library and are not explicitly represented in the graph. Therefore, all library methods are reduced to one library node that can have edges to and from application methods. The figure also shows that even for such a small sample program, the graph generated by CGC is easier to visualize and inspect. This will give the users a more focused view of the classes they are interested in, similar to what they would do during manual code inspection [15]. Having a more focused and precise view of the call graph should not ignore any of the potential call edges. Ignoring the library call-back edge in Figure 3(a), for example, will render the generated call graph unsound. Thus, it is crucial to precisely define, based on the separate compilation assumption, how the library summary node interacts with the application methods in the call graph.

3 The Separate Compilation Assumption

The input to our call graph construction algorithm is a set of Java classes designated as the *application classes*. The application classes may have dependencies on classes outside this set. We designate any class outside the set as a *library class*. We use the terms *application method* and *library method* to refer to the methods of application and library classes, respectively. The call graph construction algorithm analyzes the bytecode instructions of only the application classes. It does not analyze the instructions of any library class. However, the algorithm uses structural information (i.e., method signatures and field names) of each library class that is referenced in an application class, as well as its superclasses and superinterfaces. This is only a small subset of the full set of library classes. These referenced library classes are necessary to compile the application classes, and are readily available to the developer of the application.

The soundness of our approach depends on the **Separate Compilation Assumption**: all of the library classes are developed separately from the application classes. In particular, all of the library classes can be compiled in the absence of the application classes.

If a call graph construction algorithm does not analyze the whole program, it must make very conservative assumptions about the effects of the unanalyzed code. The separate compilation assumption makes these assumptions significantly less conservative. Without the separate compilation assumption, a sound algorithm would have to assume the following.

1. An unanalyzed class or interface may extend or implement any class or interface.

2. An unanalyzed method may instantiate an object of any type and call its constructor.
3. A local variable in an unanalyzed method may point to any object of any type consistent with its declared type.
4. A call site in an unanalyzed class may call any accessible method of any class.
5. An unanalyzed method may read or modify any accessible field of any object.
6. An unanalyzed method may read or modify any element of any array.
7. An unanalyzed method may cause the loading and static initialization (i.e., execution of the `<clinit>` method) of any class.
8. An unanalyzed method may throw any exception of any subtype of `java.lang.Throwable`.

The separate compilation assumption enables us to relieve these conservative assumptions in the following ways.

1. A library class cannot extend or implement an application class or interface. If it did, then the library class could not be compiled in the absence of the application classes.
2. An allocation site in a library method cannot instantiate an object whose run-time type is an application class. The run-time type of the object is specified in the allocation site, so compilation of the allocation site would require the presence of the application class.
   The only exception to this rule is reflective allocation sites in a library class (i.e., using `Class.forName()` and `Class.newInstance()`) that could possibly create an object of an application class. Since Java semantics do not prevent the library from doing this, our analysis should handle these reflective allocations without analyzing the library code. We assume that the library can reflectively instantiate objects of an application class if the library knows the name of this particular application class. In other words, if a string constant corresponding to the name of an application class flows to the library (possibly as an argument to a call to `Class.forName()` or `Class.newInstance()`), then the library can instantiate objects of that class.
3. Our algorithm computes a sound but non-trivial overapproximation of the abstract objects that local variables of library methods could point to. The library could create an object whose type is any library class. An object whose type is an application class can be instantiated only in an application method (except by reflection). In order for an object created in an application method to be pointed to by a local variable of a library method, an application class must pass the object to a library class in one of the following ways:
   (a) An application method may pass the object as an argument to a call of a library method. This also applies to the receiver, which is passed as the `this` parameter.
   (b) An application method called from a library method may return the object.

(c) The application code may store the object in a field that can be read by the library code.
(d) If the type of the object is a subtype of `java.lang.Throwable`, an application method may throw the object and a library method may catch it.

Thus, our algorithm computes a set, `LibraryPointsTo`, of the abstract objects allocated in the application that a local variable of a library method can point to. Implicitly, the library can point to objects whose type is any library class since these can be created in the library. Only the subset of application class objects that are passed into the library is included in `LibraryPointsTo`.

4. Two conditions are necessary in order for a call site in a library class to call a method m in an application class c.
   (a) The method m must be non-static and override a (possibly abstract) method of some library class. Each call site in Java bytecode specifies the class and method signature of the method to be called. Since the separate compilation assumption states that the library has no knowledge about the application, the specified class and method must be in the library, not the application. The Java resolution rules [22, Section 5.4.3] could change the specified class to one if its superclasses, but this must also be a library class due to the previous assumption. Therefore, the only way in which an application method could be invoked is if it is selected by dynamic dispatch. This requires the application method to be non-static and to override the method specified at the call site.
   (b) The receiver variable of the call site must point to an object of class c or a subclass of c such that calling m on that subtype resolves to the implementation in c. Therefore, an object of class c or of the appropriate subclass must be in the `LibraryPointsTo` set.
5. Similar conditions are necessary in order for a library method to read or modify a field f of an object o of class c created in the application code.
   (a) The field f must originally be declared in a library class (though c can be a subclass of that class, and could therefore be an application class). Each field access in Java bytecode specifies the class and the name of the field. This class must be a library class due to the separate compilation assumption. The Java resolution rules could change the specified class, but again only to one of its superclasses, which must also be a library class.
   (b) It must be possible for the local variable whose field is accessed to point to the object o. In other words, the `LibraryPointsTo` set must contain the abstract object representing o.
   (c) In the case of a field write, the object being stored into the field must also be pointed to by a local variable in the library. Therefore, its abstraction must be in the `LibraryPointsTo` set.

   The library can access any static field of a library class, and any field of an object that was instantiated in the library.

6. If the library has access to an array, it can access any element of it by its index. This is unlike an instance field, which is only accessible if its name is known to the library. However, the library is limited to accessing only the elements of arrays that it has a reference to (i.e., ones that are in the `LibraryPointsTo` set). In the case of a write, the object that is written into the array element must also be in the `LibraryPointsTo` set.
7. Due to the separate compilation assumption, the library does not contain any direct references to application classes. Thus the library cannot cause a static initializer of an application class to be executed except by using reflection. When determining which static initializers will execute, our algorithm includes those classes that are referenced from application methods reachable through the call graph, as well as classes that may be instantiated using reflection as discussed above in point 2.
8. The library can throw an exception either if it creates the exception object (in which case its type must be a library class) or if the exception object is created in an application class and passed into the library (in which case its abstraction appears in the `LibraryPointsTo` set). We conservatively assume that the library can catch any thrown exception. Consequently, we add the abstractions of all thrown exception objects to the `LibraryPointsTo` set.

In addition to these conditions, our algorithm strictly enforces the restrictions imposed by declared types.

- When the library calls an application method, the arguments passed in the call must be in the `LibraryPointsTo` set, and must also be compatible with the declared types of the corresponding parameters.
- When an application method calls a library method, the returned object must be in the `LibraryPointsTo` set and compatible with the declared return type of the library method.
- When the library modifies a field, the object it may write into the field must be compatible with the declared type of the field.
- When an application method catches an exception, only exceptions whose type is compatible with the declared type of the exception handler are propagated.
- When the `LibraryPointsTo` set is used to update the points-to set of an application local variable, only objects compatible with the declared type of the local variable are included.

4 CGC Overview

We have implemented a prototype of the application-only call graph construction approach that we call CGC, and have made it available at `http://plg.uwaterloo.ca/~karim/projects/cgc/`. For ease of modification and experimentation, CGC is implemented in Datalog.[3] CGC uses a pointer analysis that is

[3] Datalog is a logic-based language for (recursively) defining relations.

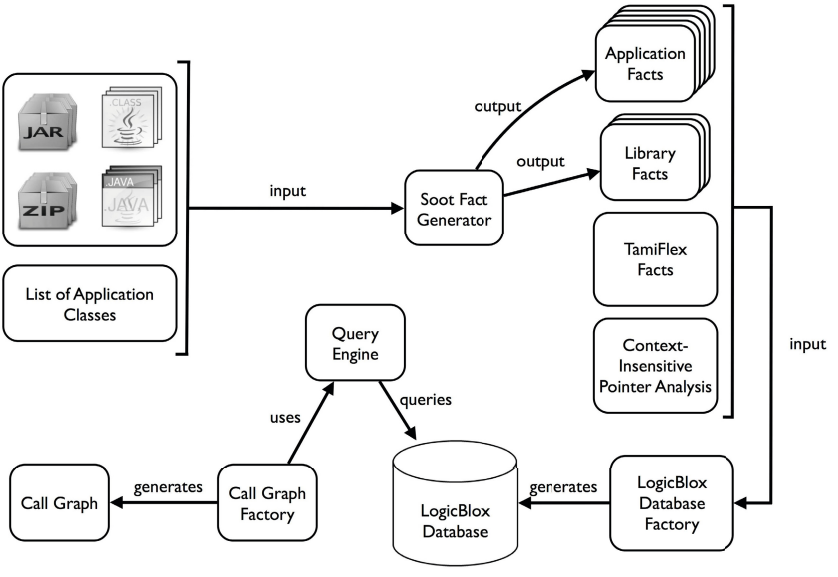


Fig. 4. An overview of the workflow of Cgc

based on the context-insensitive pointer analysis from the Doop framework [6]. However, the analysis is independent of Datalog, and could be transcribed into Java to be embedded into an analysis framework such as Soot or Wala. We have also implemented summary tools that summarize the call graphs generated by Spark and Doop. Each summary tool takes a list of application classes and the call graph as input. The output is a call graph with the library methods summarized into one node in the graph. Therefore, call graphs from Spark, Doop and Cgc can be compared. Additionally, Cgc can export the generated call graph as a GXL [16] document [4] or a directed DOT [31] graph file. The DOT graph can be visualized using Graphviz [13] or converted by Cgc to a PNG or a PS file that can be visualized using any document previewer.

4.1 Workflow

Figure 4 shows an overview of the work flow of Cgc. Like Doop, Cgc uses a fact generator based on Soot to preprocess the input code and generate the input facts to the Datalog program. The fact generator receives a collection of input files and a specification of the set of application classes. The rest of the classes are considered library classes. The fact generator then generates two sets of facts. The first set is for the application classes and contains all details about those classes: signatures for classes, methods, fields as well as facts about

[4] The DTD schema can be found at `http://plg.uwaterloo.ca/~karim/projects/cgc/schemas/callgraph.xml`

method bodies. The second set is dedicated to the library and contains the following: signatures for classes, methods, and fields in the library classes that are referenced in the application and their (transitive) superclasses and superinterfaces. We generate a third set of facts which holds information about reflection code in the application. This set is generated using TamiFlex [5], a tool suite that records actual uses of reflection during a run of a program, and summarizes them in a format suitable as input to a static analysis. CGC uses the output of TamiFlex to model calls to `java.lang.reflect.Method.invoke()`, and application class name string constants to model reflective class loading. We plan to add more support for other reflective application code in the future.

The three sets of facts along with the Datalog rules that define our pointer analysis are then used to initialize a LogicBlox [24] database. Once the database is created and the analysis completes, CGC queries it for information about the call graph entry points and the various types of call graph edges: application-to-application edges, application-to-library edges, and library-to-application call back edges. Finally, CGC uses those derived facts to generate the call graph for the given input program files and to save it as a GXL document.

4.2 Implementation

We will now outline the main parts of the CGC analysis implementation, listing the most important relations that are generated.

Object Abstraction. For precision, CGC uses a separate abstract object for each allocation site in the application classes. The abstract object represents all objects allocated at the site; it has an associated run-time type.

CGC must use a coarser abstraction to represent objects allocated in the library. CGC first computes the set L of all library classes and interfaces referenced in the application and their transitive superclasses and superinterfaces. It then creates one abstract object for each class in L.

The meaning of an abstract object in L is subtle. The abstraction must represent all objects created in the library, of any type, but L is limited to those types referenced in the application. Therefore, each abstract object $c \in L$ represents all concrete objects created in the library such that if the actual run-time type of the object is c', then c is the closest supertype of c' that is in L. In other words, each concrete object is represented by its closest supertype that is referenced in the application. From the point of view of analyzing the application, an object of type c' created in the library is treated as if its type were c. If the application accesses a field of the object, it must be a field that was already declared in c or one of its superclasses. Accessing a field declared only in c' would require the application to reference class c'. The situation is different for resolving a call to a library method as the analysis just models all library methods as a single node. Therefore, in the case of a call site in the application, the analysis does not need to determine which library method of c' or all its superclasses will be invoked.

Abstract objects allocated in the application always have a concrete class as their run-time type. An important but subtle detail is that the analysis must include abstract objects even for library classes that are declared abstract. This is because the actual run-time type of the concrete object could be a subtype of the abstract class, and not referenced by the application.

Points-to Sets. CGC uses the relations `VarPointsTo` and `StaticFieldPointsTo` to model the points-to sets of local variables and static fields within the application code, respectively. Library code cannot directly read object references from these sets. The relations `InstanceFieldPointsTo` and `ArrayIndexPointsTo` model the points-to sets of the fields of each abstract object and, in the case of an array, its array elements. The analysis distinguishes individual fields, but does not distinguish different elements of the same array.

We define a new relation, `LibraryPointsTo`, which models the set of abstract objects that the library may reference. This set is initialized with all of the abstract objects created in the library. In addition, the analysis adds abstract objects that are passed into a library method, returned to a library method from an application method, or stored in a static field of a library method. Finally, the analysis adds abstract objects that the library may read out of instance fields according to the conditions described in Section 3.

The analysis also includes rules to update `InstanceFieldPointsTo` and `ArrayIndexPointsTo` in order to model the instance field and array element writes that may occur within the library.

In addition to objects, the library can also instantiate arrays. The analysis adds to `LibraryPointsTo` an abstract array of type `T[]` whenever the application calls a library method whose return type is `T[]`, or the library calls an application method that takes `T[]` as a parameter.

Points-to Subset Constraints. CGC defines a relation called `Assign` that represents subset constraints between the points-to sets of local variables in the application code. This relation models assignment statements in reachable methods, and parameter passing and return at each method call edge. CGC defines two additional relations, `AssignToLibrary` and `AssignLibraryTo`, for assignments crossing the boundary between the application and the library. This includes parameter passing and return, as well as reading from and writing to static library fields within the application code.

For precision, we have found that it is very important to enforce declared types at the boundary between the application and the library. Since the input to CGC is Java bytecode, which is typed, there are few assignments (both intra-procedural and inter-procedural) within the application where explicit type checks are necessary (except for explicit casts in the bytecode). However, because the `LibraryPointsTo` set represents all references within the library, it does not have a declared type. Thus, at every assignment out of the library into an application local variable, instance/static field, or array element, CGC checks that the abstract objects respect the declared type of the destination.

Call Graph Edges. CGC defines the `ApplicationCallGraphEdge` relation to model calls within application methods. Additionally, two special relations, `LibraryCallGraphEdge` and `LibraryCallBackEdge` are defined to represent calls into and back out of the library. For call sites in the application, the points-to set of the receiver variable is used to resolve the dynamic dispatch and determine which methods may be called. Constructing the `LibraryCallBackEdge` set is more interesting since CGC knows nothing about the call site within the library. Following the conditions defined in Section 3, CGC uses the `LibraryPointsTo` set as an overapproximation of the possible receiver objects. CGC considers the signatures of all application methods that override a library method as possible targets for a `LibraryCallBackEdge`.

CGC computes the `Reachable` set of all methods that are transitively reachable through the call edges. The set contains application methods only.

4.3 Special Handling of `java.lang.Object`

Every constructor calls the constructors of its transitive superclasses. Therefore, every object ever created flows to the constructor of `java.lang.Object` and would be accessible to the library. However, this particular constructor is empty; it cannot leak a reference to other library code. The analysis therefore makes a special exception so that objects passed to this constructor are not added to `LibraryPointsTo`. The same exception is made for all other methods of `java.lang.Object` except `toString()`. We have determined by manual inspection that these methods do not leak object references to the library. Since the `java.lang.Object.clone()` method returns a copy if its receiver, we model it as follows: at any call site of the form `a = b.clone()`, all references pointed to by `b` flow to `a`.

5 Experiments

We evaluate CGC by comparing its precision and performance to that of SPARK and DOOP on two benchmark suites. We analyzed both the DaCapo benchmark programs, v.2006-10-MR2 [3], and the SPEC JVM 98 benchmark programs [29] with JDK 1.4 (jre1.4.2.11) which is larger than JDK 1.3 used by Lhoták and Hendren [20] and similar to JDK 1.4 used by Bravenboer and Smaragdakis [6]. We also evaluate the soundness of the call graphs generated by CGC by comparing them to the dynamic call graphs recorded at run time using the *J tool [11]. We ran all of the experiments on a machine with four dual-core AMD Opteron 2.6 GHz CPUs (runnning in 64-bit mode) and 16 GB of RAM. We exclude the benchmarks fop and eclipse from our evaluation because they do not include all code that they reference, so we were unable to analyze them with SPARK and DOOP. We exclude the benchmark jython because it heavily uses sophisticated forms of reflection, making any static analysis impractical.

5.1 Preliminaries and Experimental Setup

Since we are comparing the generated graph from the three different tools, CGC, SPARK and DOOP, we have to run them with similar settings so that the generated graphs are comparable. Therefore, there is a common properties file that holds the values for the input files, list of application classes, the benchmark to run and the name of the main class to be used across the three tools. The main class is an application class whose `main()` method is considered the entry point of the call graph.

Each experiment run is executed by a bash shell script that takes this properties file as an input. The script then runs the three tools consecutively and collects the results. For each benchmark program, the script records some statistics about the elapsed time for each tool to finish execution and the number of the various types of call graph edges generated. The script also reads in the dynamic call graph for the corresponding benchmark program to be used in evaluating the soundness of all three static analysis tools. The dynamic call graphs are generated using *J [11], a tool which attaches to the Java VM and records all method calls that occur in an actual run of the given benchmark program. The bash script also produces GXL files for the generated call graphs for CGC, SPARK, DOOP, and the dynamic call graphs. The script then computes and records the differences between them. This is done by generating four difference graphs: CGC-SPARK, SPARK-CGC, CGC-DOOP, and DOOP-CGC. A difference graph A-B is a graph that contains all of the edges that are in A and not in B.

The *J tool records a call from method a to method b if the method b ever executes on a thread during the execution of method a on the same thread. There are several situations in which the Java VM triggers such a method execution that are not due to method calls. We remove these edges from the *J call graph. First, we remove edges to the methods `java.lang.ClassLoader.loadClassInternal()` and `java.lang.ClassLoader.checkPackageAccess()`, which are called internally by the Java VM. We also remove edges to static initializers (`<clinit>`), because CGC treats static initializers as entry points, whereas *J treats them as methods that are called. Nevertheless, CGC considers static initializers as reachable methods and analyzes their effect, including any method calls that they make.

In addition, while converting SPARK's call graphs to CGC's format for comparison, we convert `NewInstanceEdges` to `LibraryCallBackEdges`. In SPARK, `NewInstanceEdges` represent implicit calls to constructors from the method `java.lang.Class.newInstance()`. In a SPARK call graph, the source of those edges is the calling site of the method `java.lang.Class.newInstance()`. On the other hand, those edges are `LibraryCallBackEdges` in CGC. Therefore, this conversion allows us to do a fair comparison between SPARK and CGC by resolving any inconsistencies in the way both model `NewInstanceEdges`.

In the following subsections, we evaluate and compare the soundness, precision, and size of the call graphs generated by the static tools, as well as the performance of the tools.

Table 1. Comparing the soundness of CGC, DOOP, and SPARK with respect to `ApplicationCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
DYNAMIC	3066	3733	482	1505	565	435	1894	2543	39	36	520	2384	5	317
DYNAMIC-CGC	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DYNAMIC-DOOP	0	0	0	325	244	193	153	250	0	0	0	0	0	0
DYNAMIC-SPARK	0	0	0	59	0	155	0	59	0	0	0	0	0	0

Table 2. Comparing the soundness of CGC, DOOP, and SPARK with respect to `LibraryCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
DYNAMIC	372	475	168	119	148	99	157	325	4	17	76	148	5	13
DYNAMIC-CGC	0	0	0	0	0	1	0	0	0	0	0	0	0	0
DYNAMIC-DOOP	0	0	0	5	53	45	43	42	0	0	0	0	0	0
DYNAMIC-SPARK	0	0	0	1	0	27	0	9	0	0	0	0	0	0

Table 3. Comparing the soundness of CGC, DOOP, and SPARK with respect to `LibraryCallBackEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
DYNAMIC	11	49	7	3	13	5	36	85	0	1	0	6	3	0
DYNAMIC-CGC	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DYNAMIC-DOOP	3	0	0	1	6	3	29	78	0	0	0	0	0	0
DYNAMIC-SPARK	0	0	0	1	4	3	3	28	0	0	0	0	0	0

5.2 Call Graph Soundness

The separate compilation assumption makes it easier to reason soundly about library code than whole-program approaches. Whereas a whole-program analysis must soundly model all details of the library, CGC needs only to soundly handle its interface. We evaluate the soundness of CGC, DOOP, and SPARK by counting the call graph edges that are present in the dynamic call graph, but missing from the call graphs generated by each static tool. These counts are shown in the lines DYNAMIC-CGC, DYNAMIC-DOOP, and DYNAMIC-SPARK in Tables 1, 2 and 3.

This comparison shows that the call graphs generated by CGC soundly include all of the call edges that were dynamically observed at run time by *J, except for one single call edge from the application to the library in the lusearch benchmark (see Table 2). The dynamic call graph for lusearch has a `LibraryCallGraphEdge` from the method `org.apache.lucene.index.FieldInfos.fieldName()` to the constructor of `java.lang.NullPointerException`. There is no statement in this method that constructs this object, but the method tries to access the field of a `null` object. As a result, the Java VM creates a `java.lang.NullPointerException` and executes its constructor. The exception is caught elsewhere in the benchmark. This type of unsoundness can be resolved by improving our analysis to model the behavior of the Java VM by checking for when an application attempts to dereference `null`. This same unsoundness is also found in both DOOP and SPARK.

Table 4. Comparing the precision of CGC with respect to `ApplicationCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	6276	14430	1879	9015	923	1998	4594	11628	40	47	653	8194	6	400
DOOP	6293	12433	1811	6245	449	1507	3956	8911	40	47	646	8188	6	400
SPARK	6299	13419	6732	7792	1412	2724	6893	13020	40	47	646	8519	6	400
CGC-DOOP	13	1997	68	2445	230	298	485	2654	0	0	7	6	0	0
CGC-SPARK	2	1829	15	1394	0	250	179	1356	0	0	7	0	0	0

Table 5. Comparing the precision of CGC with respect to `LibraryCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	649	874	571	982	243	345	431	1149	13	24	107	313	7	34
DOOP	649	841	529	787	152	251	348	818	13	24	98	313	6	34
SPARK	661	885	1060	869	336	392	797	1055	13	23	97	317	6	34
CGC-DOOP	0	33	42	190	38	50	40	289	0	0	9	0	1	0
CGC-SPARK	0	10	1	139	0	29	3	171	0	1	10	2	1	0

Table 6. Comparing the precision of CGC with respect to `LibraryCallBackEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	47	190	95	663	27	48	114	696	0	2	10	25	12	1
DOOP	39	84	55	24	15	26	37	61	0	2	0	23	8	0
SPARK	73	223	490	69	132	146	135	464	10	12	10	41	64	10
CGC-DOOP	5	106	40	638	6	19	48	557	0	0	10	2	4	1
CGC-SPARK	0	19	8	616	1	5	25	307	0	0	10	0	1	1

5.3 Call Graph Precision

In order for a call graph to be useful, it must also be precise in addition to being sound. We now compare the precision of call graphs generated by CGC to those generated by DOOP and SPARK. We would expect CGC to be at least as imprecise as DOOP and SPARK, since it makes conservative assumptions about the library code instead of precisely analyzing it. Since we found some dynamic call edges that were missing from the call graphs generated by both DOOP and SPARK (and one edge from CGC), we first correct this unsoundness by adding the missing dynamic call edges to the static call graphs. This enables us to compare the precision of the static call graphs by counting only spurious call edges, and to avoid confounding due to differences in soundness. In Tables 4, 5 and 6, the quantity CGC-DOOP represents the number of edges in the call graph generated by CGC that are missing in the call graph generated by DOOP, and are also not present in the dynamic call graph. The quantity CGC-SPARK is defined similarly. We say that CGC is *precise* when the call graph that it generates is identical to that generated through whole program analysis by DOOP or SPARK.

Application Call Graph Edges. Table 4 shows that CGC generates precise call graphs with respect to `ApplicationCallGraphEdges` when compared to both DOOP and SPARK for compress, db, jess, and raytrace. Additionally, CGC generates precise call graphs for luindex and javac when compared to SPARK. For all benchmark programs, CGC generates a median of 41 extra `ApplicationCallGraphEdges` (min: 0, max: 2656, median: 40.5) when compared to DOOP and a

median of 5 extra `ApplicationCallGraphEdges` (min: 0, max: 1829, median: 4.5) when compared to SPARK. Both medians are negligible as they represent 2.42% and 0.13% of the median number of `ApplicationCallGraphEdges` generated by DOOP and SPARK respectively.

Library Call Graph Edges. Table 5 shows that CGC generates precise call graphs with respect to `LibraryCallGraphEdges` when compared to DOOP for antlr, compress, db, javac, and raytrace. On the other hand, CGC generates precise call graphs when compared to SPARK for antlr, luindex, compress, and raytrace. Across all benchmark programs, CGC generates a median of 21 extra `LibraryCallGraphEdges` (min: 0, max: 288, median: 21) when compared to DOOP as opposed to a median of 2 extra `LibraryCallGraphEdges` (min: 0, max: 171, median: 1.5) when compared to SPARK.

Library Call Back Edges. Table 6 shows that CGC generates precise call graphs with respect to `LibraryCallBackEdges` when compared to DOOP and SPARK for compress and db. Additionally, CGC generates precise call graphs for antlr and javac when compared to SPARK. In general, CGC generates a median of 8 extra `LibraryCallBackEdges` (min: 0, max: 638, median: 8) when compared to DOOP and a median of 3 extra `LibraryCallBackEdges` (min: 0, max: 616, median: 7.5) when compared to SPARK. The former represents 72.9% as opposed to only 2.53% for the latter, of the median number of `LibraryCallBackEdges` generated by DOOP and SPARK respectively. In other words, the majority of `LibraryCallBackEdges` generated by CGC are spurious compared to DOOP. These extra edges are the root cause of the small amounts of imprecision that we observed in the `ApplicationCallGraphEdges` and `LibraryCallGraphEdges`.

We further investigate the specific causes of the extra `LibraryCallBackEdges` in the CGC call graph. In Tables 7 and 8, we categorize these edges by the name of the application method that is being called from the library. In particular, we are interested to know whether the library calls a wide variety of application methods, or whether the imprecision is limited to a small number of well-known methods, which could perhaps be handled more precisely on an individual basis.

Table 7 shows that the most frequent extra `LibraryCallBackEdges` in CGC when compared to DOOP target the commonly overridden methods (in descending order): `clone`, `<init>`, `toString`, `equals`, `remove`, and `hashCode`. The number of extra `LibraryCallBackEdges` generated by CGC compared to SPARK is smaller than that compared to DOOP. Table 8 shows that the most frequent extra `LibraryCallBackEdges` in CGC compared to SPARK target the methods: `<init>`, `finalize`, `run`, `close`, `write`, and `remove`.

The constructor `<init>` ranks highly in hsqldb, pmd, and xalan. This is because these benchmarks use class constants. CGC conservatively assumes that if a class constant is created (using `java.lang.Class.forName()`), an object of that type might also be instantiated and its constructor called.

The benchmark programs hsqldb and xalan have the highest frequency of imprecise `LibraryCallBackEdges`. In the case of hsqldb, most of those imprecise

Table 7. Frequencies of extra `LibraryCallBackEdges` in CGC when compared to DOOP (CGC-DOOP). *Other* methods include all methods that are encountered only in one benchmark program.

Method	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace	Total
clone	4	50	25		3	4		11				1			98
<init>				20			19	33							72
toString		2	2	7	2	2	2								17
equals	1		1	6		4	2	2							16
remove		12					2								14
hashCode			4	2		4	2	1							13
write				2				9							11
run				4		1		2						1	8
close				5				3							8
printStackTrace							5	2							7
next		4		1											5
getAttributes				1				3							4
getType				1				3							4
read				2								1			3
clearParameters				1				2							3
accept					1	1									2
previous		1		1											2
Other	0	37	8	585	0	3	16	486	0	0	10	0	4	0	1149
Total	5	106	40	638	6	19	48	557	0	0	10	2	4	1	1436

Table 8. Frequencies of extra `LibraryCallBackEdges` in CGC when compared to SPARK (CGC-SPARK). *Other* methods include all methods that are encountered only in one benchmark program.

Method	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace	Total
<init>				19			19	48							86
finalize				3	1	4		3							11
run				4		1		2						1	8
close				5				3							8
write				2				6							8
remove		5					2								7
getType				1				2							3
clearParameters				1				2							3
previous		1		1											2
Other	0	13	8	580	0	0	4	241	0	0	10	0	1	0	857
Total	0	19	8	616	1	5	25	307	0	0	10	0	1	1	993

`LibraryCallBackEdges` are to methods of classes in the package `org.hsqldb.jdbc` (DOOP = 560, SPARK = 555). In xalan, most of the imprecise `LibraryCallBackEdges` are to methods of classes in the packages `org.apache.xalan.*` (DOOP = 276, SPARK = 184) and `org.apache.xml.*` (DOOP = 263, SPARK = 112). Thus, in both of these benchmarks, the high imprecision is due to the fact that each benchmark contains its own implementation of a large subsystem (JDBC and XML) whose interface is defined in the library.

5.4 Call Graph Size

As we mentioned earlier, call graphs are a key prerequisite to all interprocedural analyses. Therefore, any change in the size of the call graph will affect the performance of the analyses that use it as input. Since CGC overapproximates the generated call graph, we evaluate the size of the generated call graph

Table 9. Comparing the size of the call graph generated by CGC, DOOP and SPARK for the same input program

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	6,972	15,494	2,545	10,660	1,193	2,391	5,139	13,473	53	73	770	8,532	25	435
DOOP	6,981	13,358	2,395	7,056	616	1,784	4,341	9,790	53	73	744	8,524	20	434
SPARK	7,033	14,527	8,282	8,730	1,880	3,262	7,825	14,539	63	82	753	8,877	76	444
CGC/DOOP	0.99	1.16	1.06	1.51	1.94	1.34	1.18	1.38	1	1	1.03	1	1.25	1
CGC/SPARK	0.99	1.07	0.31	1.22	0.63	0.73	0.66	0.93	0.84	0.89	1.02	0.96	0.33	0.98

(in terms of total number of edges) compared to those of DOOP and SPARK. Table 9 shows that CGC generates call graphs of equal or smaller size to DOOP and SPARK for the benchmark programs antlr, chart, compress, db, jack, javac, and raytrace. Additionaly, CGC generates call graphs of smaller size than SPARK for the benchmark programs bloat, luindex, lusearch, pmd, xalan, and jess.

It is counterintuitive that the call graphs generated by CGC are smaller than those generated by SPARK, which analyzes the whole program precisely. This result is primarily due to imprecisions in SPARK. To model objects created by the Java VM or by the Java standard library using reflection, SPARK uses special abstract objects whose type is not known (i.e., any subtype of `java.lang.Object`). SPARK does not filter these objects when enforcing declared types, so these objects pass freely through casts and pollute many points-to sets in the program. This affects the precision of the points-to sets of the method call receivers and leads to many imprecise call graph edges in SPARK. The extent of this imprecision spark was a surprise to the second author, who is also the author of SPARK. In response to this observation, we plan to improve the precision of SPARK by redesigning the mechanism that it uses to model these objects.

5.5 Analysis Performance

We evaluate the performance gain of not analyzing the method bodies of the library code. There are two major aspects that can measure the performance of an analysis: execution time, and the disk footprint of the analysis database. We define *execution time* to be the time taken by the analysis to finish computing the points-to sets and constructing the call graph for the input program. We measure this time by using the `time` Linux command [23] that measures the time taken by a given program command to finish execution. Based on the numbers shown in Figure 5, we can deduce that CGC is approximately 3.5x faster than DOOP (min: 0.86x, max: 18.42x, median: 3.45x), and almost 7x faster than SPARK (min:0.93x, max: 42.7x, median: 6.56x).

Figure 6 shows that CGC achieves this performance gain in execution time while using a database of facts approximately 7x smaller in size than DOOP (min: 2.19x, max: 31.75x, median: 6.83x). We measure the size of the database of facts used by CGC by calculating the disk footprint of the LogicBlox database files after the analysis completes. We calculate the size of the database of facts for DOOP similarly and compare it against its CGC counterpart. In all benchmarks, DOOP has a larger disk footprint as it analyzes the method bodies for all the

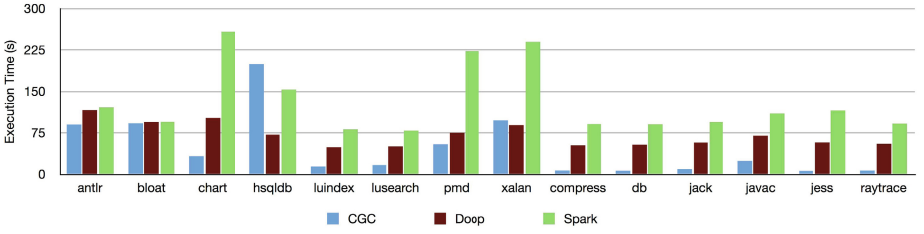


Fig. 5. Comparing the time taken by the analysis in each of CGC, DOOP and SPARK to generate the call graph for each program from the DaCapo and SPEC JVM benchmarks. This only includes the time taken to compute the points-to sets as well to construct the call graph.

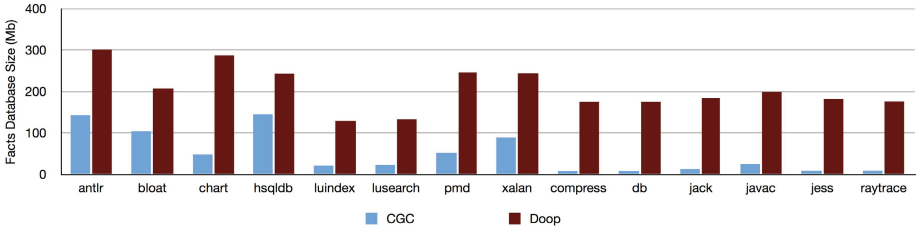


Fig. 6. The size of CGC's facts database compared to DOOP's

library classes while CGC only analyzes the method signatures for the classes referenced in the application (and their transitive superclasses and superinterfaces). It is difficult to report similar statistics for SPARK as it uses a different model to represent the facts its analysis uses to construct the call graph for an input program.

6 Related Work

Early work on call graph construction used only simple approximations of run-time types to model dynamic dispatch. Dean et al. [9] formulate *class hierarchy analysis* (CHA), which does not propagate object types. CHA uses only the subclass hierarchy to determine method targets. Bacon and Sweeney [2] define *rapid type analysis* (RTA), a refinement of CHA that considers as possible receivers only classes that are instantiated in the reachable part of the program. Sundaresan et al. [30] introduce an even more precise approach, *variable type analysis* (VTA). Like points-to analysis, it generates subset constraints and propagates points-to sets to approximate the run-time types of receivers.

Tip and Palsberg [32] provide a scalable propagation-based call graph construction algorithm. Separate object sets for methods and fields are used to approximate the run-time values of expressions. The algorithm is capable of

analyzing incomplete applications by associating a single set of objects, S_E, with the outside world (i.e., the library). The algorithm conservatively assumes that the library calls back any application method that overrides a library method. The set S_E is then used to determine the set of methods that the external code can invoke by the dynamic dispatch mechanism. A separate set of objects S_C is associated with an external (i.e., library) class if the objects passed to the methods in class C interact with other external classes in limited ways. An example of this case is the class `java.util.Vector`. This step requires the analysis of the external classes to model the separate propagation sets. In addition, this technique varies based on the library dependencies of the input program.

The previous algorithm was later used by Tip et al. [33] to implement Jax, an application extractor for Java. The external object set S_E inspired the `LibraryPointsTo` relation in our algorithm. Expanding on the initial idea of analyzing incomplete applications, we formulated the separate compilation assumption, and worked out in detail the specific assumptions flowing from it. We also derived a set of constraints from those assumptions. In addition, we have empirically analyzed the precision and soundness of the partial call graphs generated compared to call graphs generated by analyzing the whole program.

Grothoff et al. [14] present Kacheck/J, a tool that is capable of identifying accidental leaks of sensitive objects. Kacheck/J achieves that by inferring the *confinement* property [35,36,38] for Java classes. A Java class is considered *confined* when objects of its type are encapsulated in its defining package. The analysis needs only to analyze the defining package of the given Java class to infer its confinement property. That is similar to the way CGC needs only to analyze the application classes to construct its call graph. Although the set of application classes can be thought of as one defining package, determining the confinement property for the classes in this package is not enough to construct the call graph. Constructing the call graph would still require the points-to set information. As part of the confinement analysis, Kacheck/J identifies *anonymous* methods which are guaranteed not to leak a reference to their receiver. A similar notion and analysis could be used in CGC to identify library methods that do not retain permanent references to their arguments, in order to improve the precision of the `LibraryPointsTo` set. In addition to the analysis that infers the confinement property, there is a large body of work on type systems that enforce encapsulation by restricting reference aliasing [8,10,12,25].

Our work is also related to the work of Zhang and Ryder [37]. They provide a fine-tuned data reachability algorithm to resolve library call-backs, $V^a - DataReach^{ft}$. Their algorithm also distinguishes library code from application code in the formulation of the constraints. The fundamental difference is that the purpose of their algorithm is to construct a more precise call graph than a whole-program analysis by analyzing the library more thoroughly. Each call into the library is treated as an isolated context. In contrast, our aim is not to analyze the library at all, and generate a possibly less precise but sound call graph. In CGC, we make up for not analyzing the library by enforcing the restrictions that follow from the separate compilation assumption.

Rountev et al. [26] present a general approach for adapting whole-program class analyses to operate on program fragments. Whereas our aim is to analyze the application without the library, they soundly analyze the library without the application. The authors create placeholders to serve as representatives for and simulate potential effects of unknown code. The fragment class analysis then adds the placeholders to the input classes and treats the result as a complete program which can be analyzed using whole-program class analyses. Although Rountev et al. proved that their fragment class analysis is correct, the precision of the fragment analysis is variable as it significantly depends on the underlying whole-program analysis. Therefore, a CHA-dependent fragment analysis is less precise than an RTA-dependent fragment analysis. In contrast, the precision of the call graph construction in CGC depends only on the separate compilation assumption and its consequences.

DOOP [6] implements various pointer analysis algorithms for Java programs, all defined declaratively in Datalog. DOOP constructs the call graph on-the-fly while computing the points-to sets. However, there is no way to exclude some classes (e.g., the library classes) from the analysis as DOOP analyzes all of the input program. The pointer analysis in CGC is an extended version of DOOP's context-insensitive pointer analysis. The major difference is the introduction of the library summary relation and the necessary associated Datalog rules.

Lhoták and Hendren introduced SPARK [20], a flexible framework for experimenting with points-to analyses for Java programs. SPARK provides a SOOT transformation that constructs the call graph of the input program on-the-fly while calculating the points-to sets. It is possible to setup up the SOOT classes so that SPARK ignores some of the input classes. However, this is usually achieved through setting the *allow_phantom_refs* option to *true* which means that the ignored class will be completely discarded. Therefore, crucial information about the signatures of the classes, methods, and fields is lost which would render the generated call graph unsound. Thus, SPARK does not support excluding some of the input classes (e.g., the library classes) from the process of constructing the call graph despite the demand in the SOOT community [4].

WALA [17] is a static analysis library from IBM Research designed to support various pointer analysis configurations. WALA is capable of building a call graph for a program by performing pointer analysis with on-the-fly call graph construction to resolve the targets of dynamic dispatch calls. WALA provides the option of excluding some classes or packages while constructing the call graph. In fact, WALA excludes all the user-interface related packages from the Java runtime library by default when constructing a call graph. When this option is set, WALA limits the scope of its pointer analysis to the set of included classes. This ignores any effects the excluded classes might have on the calculation of the points-to sets. Therefore, the generated call graphs may be unsound and/or imprecise. Moreover, it is impossible to exclude crucial classes (e.g., `java.lang.Object`) from the analysis as this will cause an exception to be thrown. We plan to empirically compare the precision, soundness, and speed of our call graph construction algorithm with WALA in future work.

7 Conclusions and Future Work

We have proposed Cgc, a tool that generates an application-only call graph with the library code represented as one summary node. The main contributions are: (1) the separate compilation assumption that defines specific assumptions about the effects that the library code could have on the various application entities; and (2) empirically showing that the separate compilation assumption is sufficient for constructing sound (with respect to dynamic call graphs) and precise (with respect to call graphs generated by Doop and Spark) application-only call graphs. Experimental results show that not analyzing the library code does not affect the soundness of the resulting call graph. In fact, in many cases (antlr, hsqldb, luindex, lusearch, pmd, and xalan) Cgc was found to be more sound than Doop and Spark. In many cases, the call graphs generated by Cgc are almost as precise as call graphs generated by Doop, and sometimes more precise than Spark. However, when the application implements a large subsystem whose interface is defined in the library (e.g., JDBC in hsqldb and XML in xalan), Cgc loses precision compared to a whole-program analysis.

Remaining imprecisions are mainly due to objects that are passed into the library, but the library does not retain permanent references to them. This could be remedied by identifying, either manually or with an analysis, specific Java standard library methods known not to retain permanent references. Cgc would then not add objects passed into these methods to the `LibraryPointsTo` set. This approach was also suggested by Tip and Palsberg [32].

Further improvements could be achieved by defining multiple libraries that have dependencies between them. Each library will then have its own `LibraryPointsTo` set that can interact with other `LibraryPointsTo` sets or points-to sets of application entities. Although this might improve the remaining imprecision in Cgc, it requires extensive analysis for the library code to define those multiple libraries. Moreover, it might not be practical to apply this technique for user libraries as they vary greatly from one application to another.

To make the analysis more useful to users, we plan to create an Eclipse plugin that wraps our analysis and the tools we created alongside. The plugin will provide users with a suitable user interface for presenting the analysis results as well as navigating the call graph. We are also planning to embed the analysis into some of the widely used analysis frameworks such as Soot and Wala.

References

1. Agrawal, G., Li, J., Su, Q.: Evaluating a Demand Driven Technique for Call Graph Construction. In: CC 2002. LNCS, vol. 2304, pp. 29–45. Springer, Heidelberg (2002)
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 1996, pp. 324–341 (1996)

3. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA 2006, pp. 169–190 (October 2006)
4. Bodden, E.: Soot-list: Stack overflow when generating call graph (May 2011), http://www.sable.mcgill.ca/pipermail/soot-list/2008-July/001831.html
5. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: 33rd International Conference on Software Engineering, ICSE 2011, pp. 241–250 (2011)
6. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, pp. 243–262 (2009)
7. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, pp. 133–146 (1999)
8. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: 13th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 48–64 (1998)
9. Dean, J., Grove, D., Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
10. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology 4(8), 5–32 (2005)
11. Dufour, B., Hendren, L., Verbrugge, C.: *J: a tool for dynamic analysis of Java programs. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, pp. 306–307 (2003)
12. Genius, D., Trapp, M., Zimmermann, W.: An Approach to Improve Locality Using Sandwich Types. In: Leroy, X., Ohori, A. (eds.) TIC 1998. LNCS, vol. 1473, pp. 194–214. Springer, Heidelberg (1998)
13. Graphviz - Graph Visualization Software (November 2011), http://www.graphviz.org/
14. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2001, pp. 241–255 (2001)
15. Holmes, R., Notkin, D.: Identifying program, test, and environmental changes that affect behaviour. In: International Conference on Software Engineering, ICSE 2011, vol. 10 (2011)
16. Holt, R., Schürr, A., Sim, S.E., Winter, A.: Graph eXchange Language (November 2011), http://www.gupro.de/GXL/dtd/gxl-1.1.html
17. IBM: T.J. Watson Libraries for Analysis WALA (May 2011), http://wala.sourceforge.net/
18. JUnit Home Page (December 2011), http://junit.sourceforge.net
19. Lhoták, O.: Comparing call graphs. In: 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2007, pp. 37–42 (2007)

20. Lhoták, O., Hendren, L.: Scaling Java Points-to Analysis Using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
21. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. 18, 3:1–3:53 (2008)
22. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
23. Linux User's Manual: time(1) (October 2011), http://www.kernel.org/doc/man-pages/online/pages/man1/time.1.html
24. LogicBlox Home Page (November 2011), http://logicblox.com/
25. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
26. Rountev, A., Milanova, A., Ryder, B.G.: Fragment class analysis for testing of polymorphism in Java software. IEEE Trans. Softw. Eng. 30, 372–387 (2004)
27. Rountev, A., Ryder, B.G., Landi, W.: Data-flow analysis of program fragments. In: 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-7, pp. 235–252 (1999)
28. Sreedhar, V.C., Burke, M., Choi, J.D.: A framework for interprocedural optimization in the presence of dynamic class loading. In: ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 196–207 (2000)
29. Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (May 2011), http://www.spec.org/jvm98/
30. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, pp. 264–280 (2000)
31. The DOT Language (November 2011), http://www.graphviz.org/content/dot-language
32. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, pp. 281–293 (2000)
33. Tip, F., Sweeney, P.F., Laffra, C., Eisma, A., Streeter, D.: Practical extraction techniques for Java. ACM Trans. Program. Lang. Syst. 24, 625–666 (2002)
34. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
35. Vitek, J., Bokowski, B.: Confined types. In: 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 82–96 (1999)
36. Vitek, J., Bokowski, B.: Confined types in Java. Softw., Pract. Exper. 31(6), 507–532 (2001)
37. Zhang, W., Ryder, B.G.: Automatic construction of accurate application call graph with library call abstraction for Java: Research Articles. J. Softw. Maint. Evol. 19, 231–252 (2007)
38. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. J. Funct. Program. 16(1), 83–128 (2006)

Program Sliding

Ran Ettinger

IBM Research - Haifa
rane@il.ibm.com

Abstract. As program slicing is a technique for computing a subprogram that preserves a subset of the original program's functionality, program sliding is a new technique for computing two such subprograms, a slice and its complement, the co-slice. A composition of the slice and co-slice in a sequence is expected to preserve the full functionality of the original code.

The co-slice generated by sliding is designed to reuse the slice's results, correctly, in order to avoid re-computation causing excessive code duplication. By isolating coherent slices of code, making them extractable and reusable, sliding is shown to be an effective step in performing advanced code refactorings.

A practical sliding algorithm, based on the program dependence graph representation, is presented and evaluated through a manual sliding-based refactoring experiment on real Java code.

Keywords: Program slicing, sliding, co-slicing, reuse, refactoring.

1 Introduction

Program slicing, the study of meaningful subprograms that capture a subset of an existing program's behavior, can assist in building automatic tools for refactoring [4]. Slice extraction is the art of collecting a slice's set of not-necessarily contiguous program statements into a single code fragment, and reusing that fragment in the original code. With the goal of assisting programmers in maintaining high quality code, a solution to the problem of slice extraction along with its contribution to refactoring research are explored.

An advanced technique for the automation of slice extraction is introduced, through a family of code motion transformations called *sliding*. A sliding algorithm generates two subprograms, a slice and its complement, the *co-slice*, whose composition in a sequence preserves the original program's functionality. When preservation of functionality cannot be guaranteed, a sliding tool would warn the user and offer corrective measures, known in the literature as *compensation*, or compensatory code.

Deviating from earlier practices of code extraction, where the input to the transformation includes some selection of statements to be made contiguous, sliding takes a set of variable names V as input, expecting to turn the slice of code for computing the final value of V into a contiguous fragment; and

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 713–737, 2012.

while different sets of variables $V1$ and $V2$ may have the exact same slice, the co-slice computed by sliding to each set may differ. For example, in the `putInCacheIfAbsent` method shown on Fig. 1, taken from the Java compiler in Eclipse, the set of statements {1,2,6,8,9,11,13-19,21,22} (see top part of Fig. 2) forms the slice of V = {`index`}, in the scope of that method, as well as the slice of V^+ = {`index,k2V,entry,cAC,cAC.*,k2V.*`}, or the slice of any set being both a subset of V^+ and a superset of V. (Note the use of acronym of camel-case variable names, for brevity. This set V^+ will be used throughout the paper.) After sliding for V, Fig. 3 shows the result of `index` from the slice is used in the co-slice on line 23. The co-slice generated by sliding of the larger set V^+, shown in the bottom part of Fig. 2, reuses more results of the slice, such as `cAC` on line 20 and `entry`, defined on line 14 of the slice and used on line 15 of the co-slice. This reuse of local variables is enabled by moving their declaration to an outer scope. One benefit of this reuse is the reduced level of code duplication: statements 8,14,17-19,21-22 were duplicated by the sliding of V but not by that of V^+. Another advantage of the further reuse and reduced duplication is the potential reduction in need for compensation. One disadvantage, on the other hand, of this extra reuse, is the need to move local declarations, potentially requiring the renaming of those whose original name is used for other variables in the extended scope. A more significant disadvantage of the reuse of local results is its impact on the ability to reuse the slice in other parts of the program. For this to work, we need to extract that slice as a reusable method. Since Java allows only one local value to be returned from a method, some alternative compensation would be needed, making the resulting code less attractive for some applications. Sliding provides the flexibility to choose between high levels of local reuse of multiple slice results and a more straightforward global reuse of a single slice result.

A sliding algorithm, along with the details of how to compute the co-slice, is presented in Sect. 3. To demonstrate the value of this new approach to the extraction of slices, Sect. 4 describes the application of sliding to previously documented refactorings: Split Loop, Replace Temp with Query (RTwQ), and Separate Query from Modifier (SQfM). Sliding is also expected to facilitate the extraction of non-contiguous code in a general flavor of the well-known Extract Method refactoring. Such automation is crucial for enabling iterative and incremental software development [4]. It is also expected to impact on potential automation of bigger refactorings, as ambitious as Fowler and Beck's Separate Domain from Presentation or Convert Procedural Design to Objects [7].

The initial work on sliding was a theoretical pursuit. The original transformation rules have been proved correct for a simple imperative programming language restricted to assignments to primitive variables of cloneable types, sequential composition of statements, conditionals, and loops [4]. The current work presents a first practical sliding algorithm based on the program dependence graph (PDG) [6,9]. The results of a preliminary evaluation are reported in Sect. 5, and the relation to previous work is discussed in Sect. 6.

```
  /**
   * @param key1 the given declaring class name
   * @param key2 the given field name or method selector
   * @param key3 the given signature
   * @param value the new index
   * @return the given index
   */
   private int putInCacheIfAbsent(final char[] key1, final char[] key2,
                                  final char[] key3, int value) {
       int index;
 1:    HashtableOfObject key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
 2:    if (key1Value == null) {
 3:        key1Value = new HashtableOfObject();
 4:        this.methodsAndFieldsCache.put(key1, key1Value);
 5:        CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
 6:        index = -value;
 7:        key1Value.put(key2, cachedIndexEntry);
       } else {
 8:        Object key2Value = key1Value.get(key2);
 9:        if (key2Value == null) {
10:            CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
11:            index = -value;
12:            key1Value.put(key2, cachedIndexEntry);
13:        } else if (key2Value instanceof CachedIndexEntry) {
               // adding a second entry
14:            CachedIndexEntry entry = (CachedIndexEntry) key2Value;
15:            if (CharOperation.equals(key3, entry.signature)) {
16:                index = entry.index;
               } else {
17:                CharArrayCache charArrayCache = new CharArrayCache();
18:                charArrayCache.putIfAbsent(entry.signature, entry.index);
19:                index = charArrayCache.putIfAbsent(key3, value);
20:                key1Value.put(key2, charArrayCache);
               }
           } else {
21:            CharArrayCache charArrayCache = (CharArrayCache) key2Value;
22:            index = charArrayCache.putIfAbsent(key3, value);
           }
       }
23:    return index;
   }
```

Fig. 1. Example code, ahead of sliding, taken from the Eclipse Java compiler's org.eclipse.jdt.internal.compiler.codegen.ConstantPool class

The main contributions of this paper are as follows:

- Practical co-slicing and sliding algorithms suitable for the real case of (sequential) Java, building on traditional (backward, static, syntax preserving) slicing and the underlying program dependence graph representation, hence deferring the responsibility for correctness, scalability, and applicability for more languages to the slicer and the dependence graph construction mechanism.
- First evidence for the applicability of sliding for solving known refactoring techniques, including a detailed account of how developers can use sliding as a building block for performing such refactorings.
- A preliminary evaluation, having transformed a well-tested massively-used real-life Java code with no detected regression.

```
       int index;
 1:    HashtableOfObject key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
 2:    if (key1Value == null) {
 6:        index = -value;
       } else {
 8:        Object key2Value = key1Value.get(key2);
 9:        if (key2Value == null) {
11:            index = -value;
13:        } else if (key2Value instanceof CachedIndexEntry) {
               // adding a second entry
14:            ~~CachedIndexEntry~~ entry = (CachedIndexEntry) key2Value;
15:            if (CharOperation.equals(key3, entry.signature)) {
16:                index = entry.index;
               } else {
17:                ~~CharArrayCache~~ charArrayCache = new CharArrayCache();
18:                charArrayCache.putIfAbsent(entry.signature, entry.index);
19:                index = charArrayCache.putIfAbsent(key3, value);
               }
           } else {
21:            ~~CharArrayCache~~ charArrayCache = (CharArrayCache) key2Value;
22:            index = charArrayCache.putIfAbsent(key3, value);
           }
       }
 1:    ~~HashtableOfObject~~ key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
 2:    if (key1Value == null) {
 3:        key1Value = new HashtableOfObject();
 4:        this.methodsAndFieldsCache.put(key1, key1Value);
 5:        CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
 7:        key1Value.put(key2, cachedIndexEntry);
       } else {
 9:        if (key2Value == null) {
10:            CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
12:            key1Value.put(key2, cachedIndexEntry);
13:        } else if (key2Value instanceof CachedIndexEntry) {
               // adding a second entry
15:            if (CharOperation.equals(key3, entry.signature)) {
               } else {
20:                key1Value.put(key2, charArrayCache);
               }
       }
23:    return index;
```

Fig. 2. A sliding example: the slice of $V^+ = \{index, k2V, entry, cAC, cAC.*, k2V.*\}$, followed by its complement, the co-slice

2 Preliminaries

The following background on program analysis and definitions regarding the scope of the program designated for extraction and its relevant state, will be needed for the precise description of a sliding algorithm.

2.1 Control Flow Graph

The control flow graph (CFG) of a code fragment is a labeled directed graph representing the order of execution of the individual statements of the program. The CFG of the code in Fig. 1 is shown on Fig. 4. It is common to make the CFG compact by grouping nodes into basic blocks [2]. The granularity of individual statement nodes, however, is convenient for construction of the program dependence graph (PDG), as it is for slicing and sliding.

```
 1:    HashtableOfObject key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
 2:    if (key1Value == null) {
 3:        key1Value = new HashtableOfObject();
 4:        this.methodsAndFieldsCache.put(key1, key1Value);
 5:        CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
 7:        key1Value.put(key2, cachedIndexEntry);
       } else {
 8:        Object key2Value = key1Value.get(key2);
 9:        if (key2Value == null) {
10:            CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
12:            key1Value.put(key2, cachedIndexEntry);
13:        } else if (key2Value instanceof CachedIndexEntry) {
               // adding a second entry
14:            CachedIndexEntry entry = (CachedIndexEntry) key2Value;
15:            if (CharOperation.equals(key3, entry.signature)) {
               } else {
17:                CharArrayCache charArrayCache = new CharArrayCache();
18:                charArrayCache.putIfAbsent(entry.signature, entry.index);
19:                index = charArrayCache.putIfAbsent(key3, value);
20:                key1Value.put(key2, charArrayCache);
               }
           } else {
21:            CharArrayCache charArrayCache = (CharArrayCache) key2Value;
22:            index = charArrayCache.putIfAbsent(key3, value);
           }
       }
23:    return index;
```

Fig. 3. An alternative co-slice for the same slice from Fig. 2. With less reuse of local variables, it duplicates more statements (8,14,17-19,21,22). Yet it is more appropriate for some applications, as the slice can be extracted into a new method, avoiding rejection due to "ambiguous results".

A CFG has nodes N, typically with a single node $n \in N$ for representing each program statement and with two additional nodes, one for the *entry* the other for the *exit*; it has directed edges E where each edge $(m, n) \in E$ represents the direct flow of control from its source m to its target n; each node is the source of at most two edges (with `switch` statements represented as nested `if`s): the exit node has no successors, normal nodes have one successor with no label on the connecting edge, and a predicate node corresponding to a conditional or a loop statement's condition has two successors, and each of the edges is labeled T or F; however, those labels are irrelevant for slicing and will be insignificant in sliding too.

2.2 Program Scope and State

Slice extraction, in this paper, is defined to work in the scope of a given fragment of code, say S, within the body of a program's method, say M. A solution to this slice-extraction problem will compute the slice of some given set of variables, say V, with respect to S. A transformed M, resulting from the replacement of S with the sequence of the slice of S on V and its complement, should preserve the functionality of M.

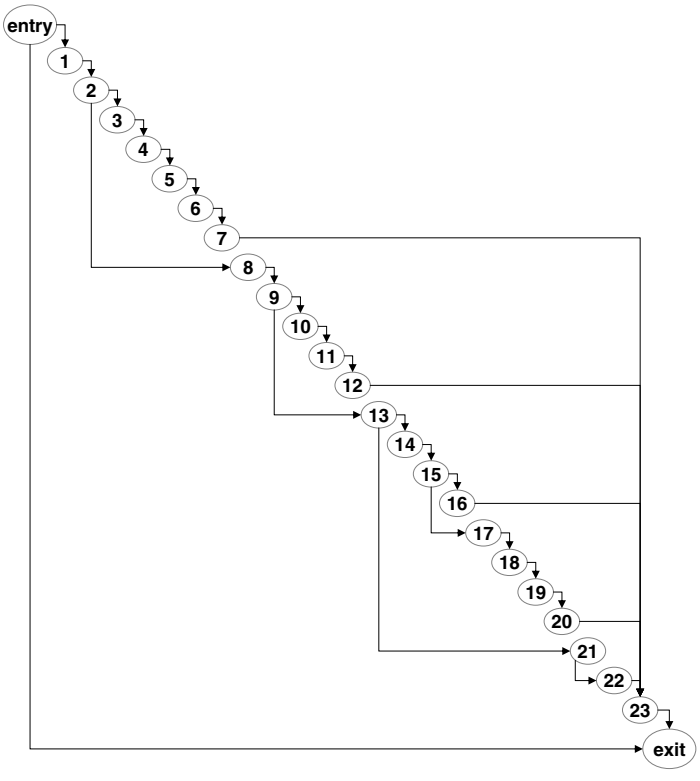


Fig. 4. A control flow graph (CFG) representation of the example code from Fig. 1. True or False labels on edges leaving predicate nodes are omitted as they are irrelevant for slicing and sliding. The entry is made into a pseudo predicate with the exit node as its other successor for convenience, making it the root of the control-dependence subgraph of the PDG.

For this transformation to be possible, the subgraph of the CFG of M corresponding to S is expected to have a single entry node and a single exit node [12,18]. This way, we can consider S as represented by its own CFG, and forget about the enclosing code in M. For such a single-entry-single-exit (SESE) region of the CFG to be extractable, a further requirement is that there is no edge from the exit node back to any node of the region.

Considering syntax, or program structure, as slicing typically generates a subprogram of the original program by deleting irrelevant statements, let's refer by the term *sub-fragment* to the result of deleting some internal statements from a given code fragment.

The set of variables each CFG node n may modify is denoted by $Def(n)$, and the set of variables it refers to is $Use(n)$. The returned value of a non-void method is given a name too, `<retval>`, and will be included in the set of defined variables whenever a `return` statement is in scope.

2.3 Background on Program Dependence

Definition 1 (Postdominance). *A node n* postdominates *a node m in a program's CFG iff every path from m to the exit includes n.*

In the example, node 20 postdominates nodes 17-20 but not node 15, due to the CFG path $< 15, 16, 23, exit >$. Node 23 postdominates all nodes except the entry.

Definition 2 (Control Dependence). *A CFG node n is* control dependent *on a CFG node m iff n postdominates a successor of m, but n does not postdominate m itself.*

Back in the example, node 20 is control dependent on node 15 because 20 is a postdominator of 17 but not of 15 itself. Note that node 20 is not control dependent on node 13, as 20 does not postdominate either successor of 13. Note also that each node that postdominates the normal successor of the entry is control dependent on the entry node, due to the special construction of the CFG's entry as a pseudo-predicate with the exit node as its other successor.

In terms of value transfer through program variables and objects, a variety of data dependence definitions exist in the literature [16,6]. Two of those, known as flow and anti dependences, will be relevant for sliding.

Definition 3 (Flow Dependence). *A CFG node n is* flow dependent *on a CFG node m iff m defines a non-empty set of variables V that are used in n, i.e. $V \subseteq \mathrm{Def}(m) \cap \mathrm{Use}(n)$, and for any $v \in V$ there exists a path from m to n in the CFG with no further definition of v.*

Definition 4 (Anti Dependence). *A CFG node n is* anti dependent *on a CFG node m iff m uses a non-empty set of variables V that are defined in n (i.e. $V \subseteq \mathrm{Use}(m) \cap \mathrm{Def}(n)$, and for any $v \in V$ there exists a path from m to n in the CFG with no other definition of v.*

We will further stress that n is flow or anti dependent on m due to variables V. This will help us decide, later on, whether to consider a dependence when computing a co-slice.

Definition 5 (PDG). *The* program dependence graph *(PDG) corresponding to a given program's CFG is a labeled directed graph with the same nodes, N, n_{entry}, n_{exit}, as in the CFG, and with an edge (m, n, tag) directed from node m to node n iff n is control or data dependent on m. The value of $Kind(tag)$ is one of $control, flow$, or $anti$. The set $Vars(tag)$ denotes the variables contributing to a (flow or anti) data dependence.*

A subset of the PDG of the example program, including all flow- and control-dependence edges, is depicted on Fig. 5. PDG-based slicing, when started with a set of nodes C, finds all nodes from which there is a directed path of flow- or control-dependence edges to any $c \in C$ [6]. A PDG-based slice starting from the

second definition of `index` at node 11, consists of the nodes $\{entry, 1, 2, 8, 9, 11\}$. One path causing the inclusion of node 1 goes through nodes 2 and 9, using a flow-dependence edge followed by two control-dependence edges.

Note that anti-dependence edges and the sets of variables causing data dependences are not used by slicing. Also, flow-dependence edges from the entry node are not significant, as the entry will be added to any non-empty slice due to control dependences. The anti-dependence edges, the labels on edges, and the flow-dependence edges from the entry node and to the exit are useful for the correct computation of a co-slice and for correctness checking. Figure 6 shows that portion of the PDG of the example code.

A traditional PDG does not include the exit node, as this node is not control dependent on any other node and does not use or define any variable. For sliding, however, it is helpful to assume $Use(exit)$ lists all variables that may be live on exit. This way, we get flow-dependence edges to the exit node from all final definition nodes of all results of the fragment represented by this PDG. A node n may define the final value of variable v if v is in $Def(n)$ and there exists a CFG path from n to the exit with no other definition of v. Since sliding extracts the slice of final values of a set of variables V, it will be convenient to start the slice from the exit node, after removing all edges to that node caused by other variables. (Similarly, a co-slice will be computed by slicing from the exit node after removing other edges.) For this to work, we must include all extracted variables V in $Use(exit)$, as well as all side effects on fields of objects that may be used outside. In Fig. 6, the local variables and object references in the example extracted sets V and V^+ are listed on the labels of flow-dependence edges to the exit node as optional, in square brackets, as they would not occur on the original PDG if it were not for the extraction.

A flow-dependence edge from the entry to the exit node lists all live-on-exit and extracted variables that may be modified but may also keep their initial value due to a CFG path from entry to exit with no definition of that variable. In the example, it is interesting to see that `index` is excluded from that edge, as its 5 definition nodes cover each potential path.

Definition 6 (Final-Use Node). *Given a code fragment S and a variable v, a node n on the CFG of S is a* final-use node *for v in S iff v is used in n and each path in the CFG from n to the exit is free of definitions of v.*

Equivalently, note that in terms of a PDG with nodes N and edges E, including anti-dependence edges, a node n is a final-use node of v if and only if $v \in Use(n)$ and there exists no anti-dependence edge $(n, n', tag) \in E$ due to v, i.e. with $Kind(tag) = anti$ and $v \in Vars(tag)$.

3 PDG-Based Sliding

3.1 Source Code Considerations

Given a code fragment S and a set of variables V, a sliding transformation replaces S with a sequence of two sub-fragments, S_V and S_{CoV}, corresponding

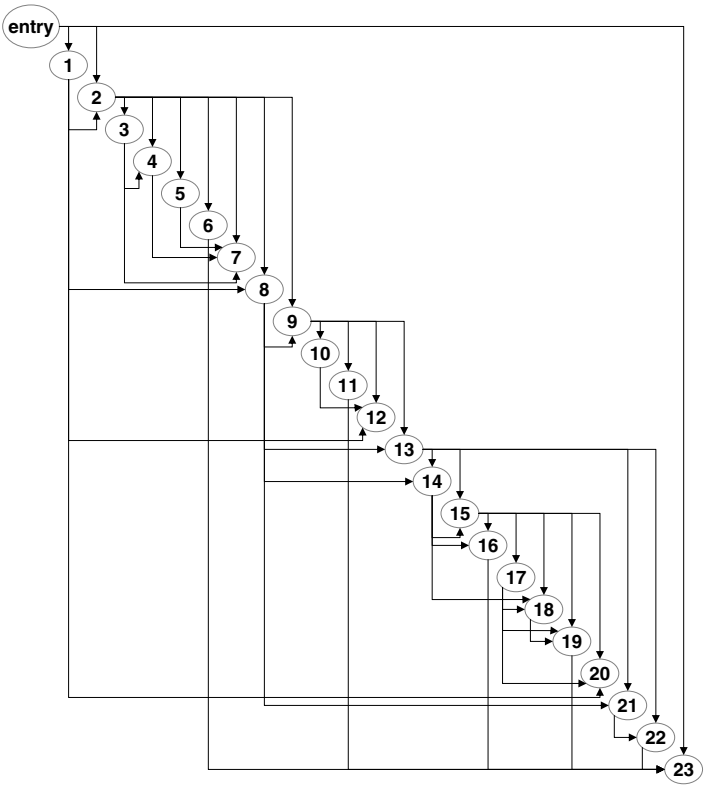


Fig. 5. A program dependence graph (PDG) representation of the example code from Fig. 1, with control- and data-dependence edges shown above and below the nodes, respectively

to the slice and co-slice of S on V, respectively. Suppose the fragment S is represented by a PDG, $(N, E, n_{entry}, n_{exit})$, as defined above. The algorithm in Fig. 7 accepts that PDG as input, along with the set of variables for extraction, V. It computes and returns a subset of nodes $N_V \subseteq N$ representing the slice, and another subset $N_{CoV} \subseteq N$ for the co-slice.

Sub-fragments S_V and S_{CoV} of S can be generated by removing all statements whose corresponding nodes are not present in N_V or N_{CoV}, respectively. We consider the removal of statements from blocks of code, even if nested in conditionals or loops. A more advanced sliding tool, supporting the decomposition of single statements too, would need to consider syntactic difficulties, and is left for future work. (For example, if the two side effects on parameters in `f(y++,z++);` are in the slice but the method call not, an extra semicolon would need to be added.) Two other syntactic considerations in the construction of S_V and S_{CoV} are related to local variable declarations and labeled blocks of statements. Those two constructs are not represented in the PDG by nodes. For the former, all declarations of vaiables not occuring in each side should be removed,

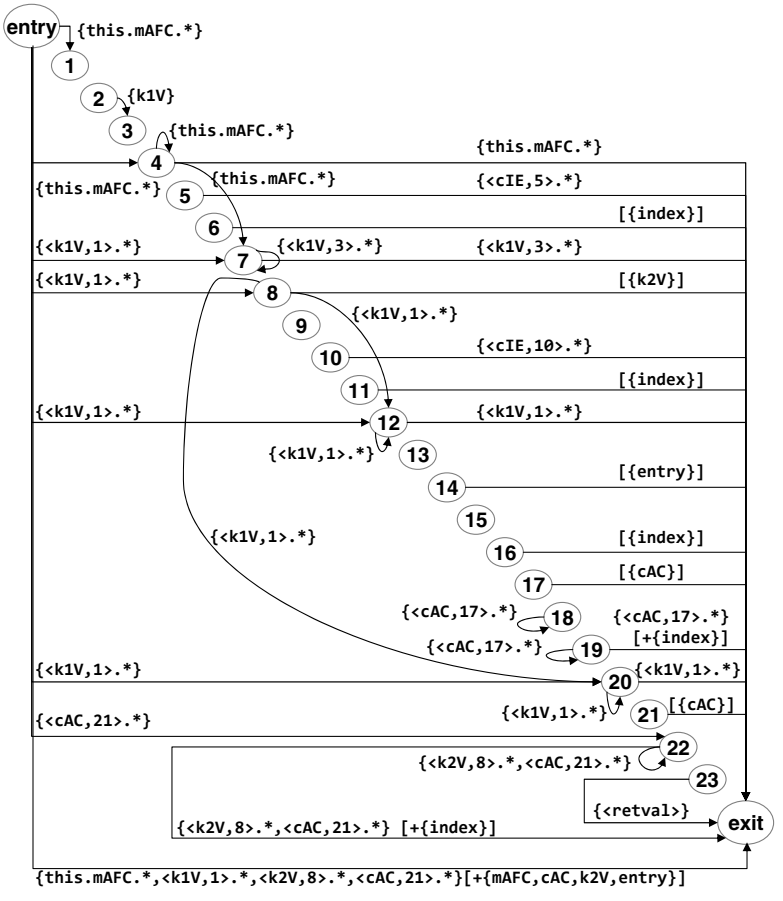


Fig. 6. A subset of the program dependence graph (PDG) representation of the example code from Fig. 1, showing flow-dependence edges from the entry node, to the exit node, as well as anti-dependence (curved) edges

and if a non-nested declaration occurs on both sides, its declaration should be removed from the co-slice, to avoid a compilation error. (An example of such declaration removal is shown on line 1 of Fig. 3). For the latter, if a non-nested block gets duplicated, its label should be renamed at least in one side.

When the functionality of the sequence of slice and its complement, in terms of the relation between input and output values for all live-on-exit variables, is guaranteed to be equivalent to that of the original code fragment, the successful sliding is considered a *compensation free* transformation. Otherwise, a variety of compensatory measures can be taken, e.g. in the form of variable localization and renaming. For this purpose, the PDG-based sliding algorithm of Fig. 7 computes and returns three further results, for the sets of potentially problematic variable instances $Pen1$, $Pen2$ and $Pen3$. Those instances may require compensation due

$ProgramSlidingOnThePDG(N, E, n_{entry}, n_{exit}, V)$
1 initialize the set of edges $FlowToExit_{NonV}$ to the empty set
2 **forall** $(m, n_{exit}, tag) \in E$ **do**
3 **if** $Vars(tag) \cap V = \emptyset$
4 add the edge (m, n_{exit}, tag) to $FlowToExit_{NonV}$
5 $N_V := ComputeSlice(N, E \setminus FlowToExit_{NonV}, n_{exit})$
6 initialize $NonFinalUsesAtNode$ to map each $n \in N$ to the empty set (of variables)
7 **forall** $(m, n, tag) \in E$ **do**
8 **if** $Kind(tag)$ is *anti*
9 add all variables in $Vars(tag)$ to $NonFinalUsesAtNode(m)$
10 initialize the set of edges $FlowToFinalUse_V$ to the empty set
11 **forall** $(m, n, tag) \in E$ **do**
12 **if** $Kind(tag)$ is *flow* and $Vars(tag) \subseteq (V \setminus NonFinalUsesAtNode(n))$
13 add (m, n, tag) to the set of final-use edges $FlowToFinalUse_V$
14 $N_{CoV} := ComputeSlice(N, E \setminus FlowToFinalUse_V, n_{exit})$
15 $(Pen1, Pen2, Pen3) := CollectVariablesRequiringCompensation(N, E, V,$
16 $n_{entry}, N_V, N_{CoV}, NonFinalUsesAtNode)$
17 **return** $(N_V, N_{CoV}, Pen1, Pen2, Pen3)$

Fig. 7. A PDG-based sliding algorithm, collecting all nodes in the selected fragment's PDG, $(N, E, n_{entry}, n_{exit})$, belonging to the slice and the co-slice of the selected set of variables V. Three sets of variable instances potentially requiring compensation to avoid unintended data flow are computed too.

to definition of variables from V in the co-slice, non-final use of such variables in the co-slice, or definition in the slice of other (non-V) variables whose initial value may be needed in the co-slice. Some approaches to overcome the potential change in functionality are discussed in Sect. 3.4.

3.2 Computing the Slice

The proposed sliding algorithm works in three steps, for computing the slice, its complement, and finally checking correctness. The first step, on lines 1-5 of Fig. 7, computes the slice of final value of variables V, by starting from the exit node after having removed all flow-dependence edges (in the set $FlowToExit_{NonV}$) coming into the exit node due to variables not in V. In the example of $V = \{\texttt{index}\}$, the only remaining edges to the exit are the edges from the five definitions of `index`, on nodes {6,11,16,19,22}. Line 5 invokes the slicing algorithm of Fig. 8 on the exit node and the remaining edges, such that the first element n on line 4 of that algorithm will be the exit node, and all its remaining predecessors {6,11,16,19,22} will be added to *Slice* on the loop of lines 5-7, causing later addition of all nodes contributing to the final value of `index`. This way, the computation of S_V can be considered as having two substeps: first adding all predecessors of the exit node due to any member of V, then repeatedly adding all other nodes with a PDG path to those. In the example, the former substep operates on the PDG subgraph shown on Fig. 6, whereas the latter substep continues using the portion of the PDG shown on Fig. 5.

```
ComputeSlice(N, E, c)
1  initialize the result set of nodes Slice to the singleton set {c}
2  similarly initialize Worklist to {c}
3  while Worklist is not empty do
4    take the first element n out of the Worklist
5    forall m ∈ N such that (m, n, tag) ∈ E do
6      if Kind(tag) is flow or control and m ∉ Slice
7        add the PDG predecessor node m of n to Slice and to Worklist
8  return Slice
```

Fig. 8. A traditional PDG-based slicing algorithm to collect all statement nodes from which there exists a directed path of dependence edges (flow or control, not anti) in the PDG, (N, E), ending in a node of interest c, known as the slicing criterion

To verify that the computed slices in the first two sliding examples are identical as expected, note that when sliding for the larger set V^+, compared with the sliding for V, further edges – such as $(8, exit)$ due to `entry` – survive the removal of $FlowToExit_{NonV}$. Still, since the source nodes of all those further edges are in the slice of `index`, these extra edges do not make the slice of V^+ larger or different. Indeed, in sliding for `index`, the edge $(8, exit)$ is removed from the PDG available for the slicing algorithm, but node 8 finds another way into that slice too. Since the edge $(11, exit)$ is not removed there, node 8 will be added to that slice due to, for instance, the dependence path $< 8, 9, 11 >$ (see Fig.5).

3.3 Computing the Co-slice

The next step, on lines 6-14 of Fig.7, is co-slicing. It involves slicing from the exit node again (line 14), but for variables outside the set V this time. To maximize reuse, the removal of redundant flow-dependence edges (lines 10-13) is not restricted to edges leading to the exit node. A flow-dependence edge $(l, m, tag) \in E$ can be removed if the variables causing it, $Vars(tag)$, form a subset of V, and the node it ends in, m is a final-use node for all those variables. Node m is guaranteed to be a final-use node of variable v if v does not contribute to any anti-dependence edge from m, i.e. for any $(m, n, tag') \in E$, we get $v \notin Vars(tag')$.

Since the sets of final-use variables at each node will be needed later in checking for correctness (lines 15-16 and the called algorithm of Fig. 9), a separate stage (lines 6-9) computes and stores this information for all nodes. For convenience (and efficiency), it maps each node to the set of *non*-final uses; it first assumes no variable is a non-final use at all nodes (line 6) and then whenever evidence is found for a non-final use, through an anti-dependence edge from a node m, it adds to the set of non-final uses of m (on line 9) all the variables this edge is due to. Notably, the use of variables on the exit node is certainly a use of their final value; this is confirmed by the lack of anti-dependence edges from the exit.

In the example, when preparing to compute the co-slice of V, all edges from the definition nodes of `index`, being from $\{6, 11, 16, 19, 22\}$ to the return statement's node, 23, are added to $FlowToFinalUse_V$ (on line 13) and hence removed from the co-slice calculation. This is so because none of those definition nodes can be reached (in terms of control flow) from the return node. Accordingly, there exists no anti-dependence edge leaving node 23 (see Fig. 6), so we never get to line 9 of the algorithm with m being 23, hence the set $NonFinalUsesAtNode(23)$ remains empty (as initialized on line 6). More tricky are the edges from those final-definitions of `index`, nodes $\{6, 11, 16, 19, 22\}$, to the exit node. The edges from the last two, $(19, exit)$ and $(22, exit)$ are due to the sets `{<cAC,17>.*,index}` and `{<k2V,8>.*,<cAC,21>.*,index}`, respectively (Fig. 6). Since those are not subsets of the set V (i.e. the singleton `{index}`), line 13 of the algorithm is not reached, and those two edges are not removed. Indeed the co-slice of V, on Fig. 3, includes statements 19 and 22 due to their side effects on the state of the objects on which the method `putIfAbsent()` is invoked. When sliding for V^+, the object fields `{<cAC,17>.*,<k2V,8>.*,<cAC,21>.*}` are included in the set of extracted variables, and hence the edges $(19, exit)$ and $(22, exit)$ are added to $FlowToFinalUse_V$ (on line 13) such that nodes 19 and 22 are not added to the co-slice (see bottom part of Fig. 2).

Note that had node 19 been added to the co-slice of V^+, node 18 would have been added too. The flow-dependence edge from node 18 to 19 (due to `<cAC,17>.*` which is in V^+) is not added to $FlowToFinalUse_V$ (on line 13), because of the anti-dependence edge $(19, 19)$, which is due to `<cAC,17>.*` (Fig. 6). In contrast to the redundancy of self dependence edges in the context of slicing, this example shows how self anti-dependence edges are relevant for co-slicing. They reflect the fact that a variable is both used and defined in a single statement, such that its value on exit from the statement may differ from its value on entry.

3.4 Correctness Checking and Compensation

The final step of the proposed PDG-based sliding algorithm involves checking for correctness of the transformation, in terms of preserving functionality. In Fig. 7, the checking procedure is invoked on lines 15-16. This procedure, detailed on Fig. 9, involves the collection of three sets of potentially problematic variable instances. When all three sets are empty, the replacement of the original code fragment with the computed slice followed by the co-slice, is guaranteed to preserve the original functionality, such that for a given input state on which the original fragment terminates, the transformed fragment would terminate too, leaving the program in the same state as the original in all variables.

Let's examine the variables for extraction V first. The responsibility for computing the correct value for those variables lies on the slicing algorithm and the input PDG it is computed on. Assuming the slice computes the expected value for each $v \in V$, we need to make sure this correct value is maintained by the co-slice. The simplest way to ensure this is to check that v is not in the set of defined variables of any node of the co-slice, $n \in N_{CoV}$. The set $Pen1$ collects

those problematic variable instances (lines 3-4). In the example of Fig. 2 this set is empty. In the example of Fig. 3, in turn, we get two instances of `index` in this set, for its definitions in nodes 19 and 22. These nodes are included in the co-slice not due to modifying `index`, but rather due to relevant side effects on live-on-exit object fields.

In terms of compensation in the co-slice, our choice on Fig. 3 was to remove the impact on the offending member of V, `index`, by removing the assignments to it. This was simple to do in this case but may be more challenging in others. For example, if the undesired definition takes place in a called method, changing its code may require the duplication of that method, in case its original version is still called from elsewhere. The flavor of sliding proposed in this paper is restricted to a local transformation of the selected fragment. Instead, if the value of that variable is accessible at the beginning of the transformed code fragment, it can be backed up in a local variable, ahead of the slice, and restored ahead of the co-slice. (Accessibility may be problematic, for example, when the variable is a private field of a different class.)

When the unwanted definition is performed in the selected fragment and is tricky to eliminate, say due to syntactic difficulties (as mentioned earlier), a preferred alternative would be to localize the effect by adding a new variable and updating the problematic instances to refer to the new variable. To enable this corrective measure, pairs of node and variable name are collected and returned by the checking procedure, instead of just variable names.

Moving on to examine all the other variables relevant (i.e. live) on exit from the original fragment, note that, by construction, the live-on-exit variables are $Use(n_{exit})$. We therefore need to consider variables in the set $CoV := Use(n_{exit}) \setminus V$.

The co-slice is computed as a slice of final values of variables CoV, using final values of V where it is guaranteed to be correct to do so. On entry to the co-slice, we can assume to have the expected final value of each variable $v \in V$ available. For correct operation of that co-slice, we first need to ensure no other (i.e non-final) value of a variable in V is used in any node $n \in N_{CoV}$. Such instances are collected in the set $Pen2$ (lines 5-6). In both earlier examples, all uses of variables V and V^+ are final. Sliding for the local object reference `key1Value`, whose slice consists of nodes 1,2, and 3, would yield a co-slice with a non-final use of that variable on node 2 – evidenced by the anti-dependence edge $(2, 3)$ on Fig. 6.

In terms of compensation, the non-final use of a variable v requires the addition of a new variable, v', and renaming of those instances of the non-final use, in the co-slice. When the initial value of v is needed in the co-slice, it should be backed up ahead of the slice, if possible. Again, the preparation of such a backup might not be possible if the variable is not local and not accessible at the head of the transformed fragment. Also, renaming might not be possible if v is not local and used in a called method. (Note that the set of variable instances $Pen2$ will specify a node n where the non-final use took place, but this may be a node involving a method call, and v may be an object field referred to inside

this method, when the object reference is passed explicitly as a parameter, not the field v itself.)

In the example of sliding for `key1Value`, were node 2 of the co-slice would include a non-final use of that variable, localization would work well, with no neeed for backup of the initial value, due to the initialization of `key1Value` on the statement of node 1 (see Fig. 1).

Finally, for all other initial values demanded by the co-slice, outside of the set V, we must ensure they hold their initial value on entry to the co-slice. We first collect all non-V variables that may require an initial value at the co-slice, in the set $InputToCoSliceNonV$ (lines 7-11). Those are the non-V variables contributing to a flow-dependence edge from the entry to any node $n \in N_{CoV}$. Then, the simplest way to ensure those variables hold the initial value on entry to the co-slice is by checking that they are not defined in the slice nodes N_V. Definitions of all members of $InputToCoSliceNonV$ in any slice node are recorded in the set $Pen3$ (lines 12-14). In the example of Fig. 3, for sliding `index`, we get the input to the duplicated node 22 in the co-slice, `<cAC,21>.*`, also defined in the slice on that same node (see labels on flow-dependence edges $(entry, 22)$ and $(22, exit)$ on Fig. 6).

A preferred form of compensation, again, would be to localize the unwanted effects in the slice, if possible. When the initial value is not needed there, and the definition is explicit (i.e. not performed inside a called method), this would be a simple addition of a local variable in the slice.

In the mentioned example, the effects on fields of the object referred to by the local variable `cAC` (i.e. `charArrayCache` in Fig. 3), are due to implicit definitions on node 22. The actual definitions take place in the method `putIfAbsent()` of the class `CharArrayCache`, so simple renaming better be avoided as it would require duplication of that method. Now, in this case the initial value of those fields is needed for correct operation of the slice, as can be witnessed by the edge $(entry, 22)$ on Fig. 6. And as it turns out, it is possible to back up those fields, since they are not private and the classes are in the same package. Still, it is not clear that this would be an acceptable practice. Alternatively, one could consider making a backup of the entire object referred to by `charArrayCache`. This would require the object to be cloneable, and it is arguable that this kind of compensation would be acceptable too. Note that such a backup would not be trivial to prepare ahead of the slice, as the local object reference `charArrayCache` is computed in the slice itself.

The measures of compensation taken in the experiment of Sect. 5 below involve variable localization and local backups, but no cloning of objects.

4 Sliding-Based Refactoring

When applied to source code, sliding can be considered a refactoring by itself, as it may improve the structure of existing code, making it more readable and easier to maintain. Naturally, a sliding transformation can be followed up by steps of method extraction, as the slice and its complement are made contiguous and hence ready for extraction by the Extract Method refactoring [7].

```
CollectVariablesRequiringCompensation(N, E, V,
      n_entry, N_V, N_CoV, NonFinalUsesAtNode)
1   initialize Pen1,Pen2, and Pen3 to empty sets
2   forall n ∈ N_CoV do
3     forall v ∈ Def(n) ∩ V do
4       add the pair (n, v) to Pen1
5     forall v ∈ Use(n) ∩ V ∩ NonFinalUsesAtNode(n) do
6       add the pair (n, v) to Pen2
7   initialize InputToCoSliceNonV to the empty set
8   forall n ∈ N_CoV do
9     forall (n_entry, n, tag) ∈ E do
10      if Kind(tag) is flow
11        add all variables in Vars(tag) \ V to InputToCoSliceNonV
12  forall n ∈ N_V do
13    forall v ∈ Def(n) ∩ InputToCoSliceNonV do
14      add the pair (n, v) to Pen3
15  return (Pen1, Pen2, Pen3)
```

Fig. 9. Analysis of the slice and co-slice to identify potential unintended flow of data if the sequence of slice and co-slice were to replace the original code

This section revisits three refactoring techniques from existing catalogs [7,1]: Split Loop, Replace Temp with Query, and Separate Query from Modifier. A revision of the mechanical description of how to perform the refactoring transformation is proposed, with sliding being a key step in all three cases. The revised mechanics, being more concrete and constructive than the original descriptions, contribute to the applicability of the refactorings, making them more amenable for future automation.

A direct application of sliding is the Split Loop refactoring [1]. Fig. 10 shows its proposed mechanics. The user can simply follow the sliding algorithm, or use a tool, and select the variables of interest for extraction. If the initialization of the loop index is not in the selected code fragment, a tool based on the compensation-free flavor of sliding, expecting empty sets in the resulting $Pen1$,$Pen2$, and $Pen3$ sets (see Fig. 9), would correctly reject the transformation, as the loop in the co-slice would be skipped.

One of the more advanced versions described above would add a backup variable for the initial value of that loop index. Another advanced sliding implementation would avoid some code duplication by identifying all final values of the loop whose computation is fully included in the extracted slice. The addition of those variables to the set of variables for extraction, V, would cause the addition of further edges to the set $FlowToFinalUse_V$ (see lines 10-13 in the sliding algorithm of Fig. 7). Those added flow-dependence edges, directed from final-value definition nodes to the exit, are then removed from consideration when computing the co-slice, potentially making the co-slice (i.e. second instance of the loop) smaller. The value of a smaller co-slice is not only in the reduced levels

Mechanics for Split Loop

- Perform sliding on the loop fragment choosing a subset V of the loop's results.
  - *Avoid unnecessary code duplication by adding to V all loop results whose addition will not increase the size of the first resulting loop.*
  - *If sliding fails update the code to avoid the failure and repeat this step, or choose a different loop to split.*
- Compile and test.

Fig. 10. Sliding-based mechanics for Split Loop

of code duplication, but also in the potential need for less compensation, as can be witnessed by the collection of $Pen1$ and $Pen2$ on lines 2-6 of Fig. 9.

New mechanics for the Replace Temp with Query (RTwQ) refactoring [7] are presented on Fig. 11. This refactoring involves the extraction of the computation of a single variable, a temporary one, into a method of its own. That method should have no side effects and it will be invoked from all places in the original code where the final value of the temp was used. Applying RTwQ twice, on the variables `def` and `pot` of Fig. 12, would replace their final use in line 2009 with calls to the new methods, on line 1995 of the resulting code, on Fig. 13. There, the two new methods, `def()` and `pot()`, can be seen starting on lines 2001 and 2016, respectively.

Note that the loop has been eliminated from the original code, in this example. Interestingly, one more loop in the same original method included the same computation of `def` and `pot`. In that other loop (not shown here), the two computations were entangled with a computation of one other value (`nullS`, whose final use can be seen on line 1983 on Fig. 13. Subsequent applications of RTwQ on that other loop, for `def` and `pot` again, would replace their final use with calls to the previously extracted methods. Those two invocations can be seen on line 1981 of the resulting code on Fig. 13. This is an example where the initial RTwQ caused duplication of the loop, yet it has enabled further refactoring steps to reduce duplication through the elimination of non-trivial code clones.

A third refactoring that can benefit from sliding is Separate Query from Modifier (SQfM) [7]. This refactoring involves the splitting of a non-void method with side effects to two methods. Like in RTwQ, the extracted slice is of a single value, the one returned from the original method. Note that this value will not necessarily reside in a single variable since the result of some expression could be returned, and since multiple return statements may refer to different variables. Accordingly, an optional first step, in preparation for sliding, is dedicated to the introduction of a local variable to hold the returned result. When the code includes more than one return statement, each return is replaced by an assignment to the added variable and a jump to the end, where a single return of that variable is inserted. In Java, the jump can be implemented through a break from

Mechanics for Replace Temp with Query

- Identify the relevant fragment of code, from the temp's declaration to the end of its enclosing block.
- Perform sliding on that code fragment for the selected temp.
  - *If the sliding fails you may want to choose a different temp to replace with a query.*
  - *Successful sliding will bring together the code for computing the final value of the temp, making it a contiguous fragment ready to be extracted into a method of its own.*
- Compile and test.
- Perform Extract Method on the extracted slice, giving it an appropriate name.
  - *If the extracted method appears to have side effects consider extracting a different temp, or modify the code to prevent the side effects.*
  - *If all those side effects occur in invoked methods, consider applying Separate Query from Modifier on those methods before re-applying this refactoring. That way, the extracted modifiers could possibly be excluded from the query, and cause no side effects.*
- Compile and test.
- Perform Inline Temp on the selected temp at its declaration.
  - *This step involves the replacement of all references to the temp with the call to the extracted method (i.e. the query), and the removal of the temp's declaration.*
  - *If the value of any of the query's parameters may be different at any point of reference, consider adding backup variables at the temp's point of declaration, or abandon the refactoring.*
  - *A potential cause of failure is the need of backup for a non-cloneable parameter object; making such backup might otherwise not be desirable due to space (i.e. large object to clone) or other considerations.*
- Compile and test.

Fig. 11. Sliding-based mechanics for Replace Temp with Query

```
1995            String def = "FlowInfo<def:[" + this.definiteInits, //$NON-
1996                pot = "], pot:[" + this.potentialInits; //$NON-NLS-1$
1997            int i, ceil;
1998            for (i = 0, ceil = this.extra[0].length > 3 ?
1999                               3 :
2000                               this.extra[0].length;
2001                i < ceil; i++) {
2002                def += "," + this.extra[0][i]; //$NON-NLS-1$
2003                pot += "," + this.extra[1][i]; //$NON-NLS-1$
2004            }
2005            if (ceil < this.extra[0].length) {
2006                def += ",..."; //$NON-NLS-1$
2007                pot += ",..."; //$NON-NLS-1$
2008            }
2009            return def + pot
2010                + "], reachable:" + ((this.tagBits & UNREACHABLE) == 0)
2011                + ", no null info>"; //$NON-NLS-1$
```

Fig. 12. Example RTwQ, original code

a labeled block. This preparatory step should be undone before the end of the refactoring, when the labeled block and result variable are located in designated methods for the query and modifier, and can be replaced with return statements.

```
1981            return def() + pot()
1982                + "], reachable:" + ((this.tagBits & UNREACHABLE) == 0)
1983                + nullS
1984                + "]>"; //$NON-NLS-1$
1985        }
1986    }
1987    else {
1988        if (this.extra == null) {
1989            return "FlowInfo<def: " + this.definiteInits //$NON-NLS-1$
1990                +", pot: " + this.potentialInits  //$NON-NLS-1$
1991                + ", reachable:" + ((this.tagBits & UNREACHABLE) == 0) ,
1992                +", no null info>"; //$NON-NLS-1$
1993        }
1994        else {
1995            return def() + pot()
1996                + "], reachable:" + ((this.tagBits & UNREACHABLE) == 0)
1997                + ", no null info>"; //$NON-NLS-1$
1998        }
1999    }
2000 }
2001 private String def() {
2002     String def = "FlowInfo<def:[" + this.definiteInits; //$NON-
2003     int i;
2004     int ceil;
2005     for (i = 0, ceil = this.extra[0].length > 3 ?
2006                        3 :
2007                        this.extra[0].length;
2008         i < ceil; i++) {
2009         def += "," + this.extra[0][i]; //$NON-NLS-1$
2010     }
2011     if (ceil < this.extra[0].length) {
2012         def += ",..."; //$NON-NLS-1$
2013     }
2014     return def;
2015 }
2016 private String pot() {
2017     String  pot = "], pot:[" + this.potentialInits;
2018     int i;
2019     int ceil;
2020     for (i = 0, ceil = this.extra[0].length > 3 ?
2021                         3 :
2022                         this.extra[0].length;
2023         i < ceil; i++) {
2024         pot += "," + this.extra[1][i]; //$NON-NLS-1$
2025     }
2026     if (ceil < this.extra[0].length) {
2027         pot += ",..."; //$NON-NLS-1$
2028     }
2029     return pot;
2030 }
```

Fig. 13. Example RTwQ, after replacement of two temporary variables with queries

After sliding the computation of this returned value away from the remaining computations (i.e. the code with the side effects), we perform Extract Method twice, on the slice and co-slice, followed by the optional undoing of the preparatory step. In the final step, we perform Inline Method, to replace all calls to the original method with the two invocations, of the query and modifier.

Beyond its immediate application as a refactoring, where the transformed code is a candidate for further development and therefore might be committed at a subsequent code change delivery, the SQfM transformation can be useful also for temporarily updating the code in a testing, debugging, or verification scenario. Such usage of SQfM might enable the application of tools and techniques that assume no side effects exist in conditional expressions.

Proposed mechanics for SQfM are presented in Fig. 14. The sliding of `index` on the code from Fig. 1, yielding the slice shown on the top of Fig. 2 and the

Mechanics for Separate Query from Modifier

- Prepare the method for sliding by ensuring it has a single return statement.
  - *Enclose all statements in the method body in a labeled block.*
  - *Insert a declaration above the added block for a new temporary variable to store the method's result, and a statement to return its final value below that block.*
  - *Modify all return statements in the labeled block to store the returned value and to break to the added label (i.e. out of the method's body through the added return statement).*
  - *This preparatory step may be skipped when the original method is already constructed with a single retrun of a single variable.*
- Compile and test.
- Perform sliding on the labeled block for the temp designated in the first step above.
  - *If sliding fails choose a different method to separate.*
- Compile and test.
- To construct the query, perform Extract Method on the slice, giving it an appropriate name.
  - *If the slice appears to have side effects consider the separation of a different method, or modify the code to prevent the side effects.*
  - *If all those side effects occur in invoked methods consider applying this refactoring on those methods before re-applying it on the present method. That way, the extracted modifiers could possibly be excluded from the query, and cause no side effects.*
- Compile and test.
- Perform Inline Temp on the designated temp if you prefer to have the query called after the modifier, possibly also inside the modifier if its result is used there.
- To construct the modifier, perform Extract Method on the updated co-slice, giving it an appropriate name.
- In the query and modifier methods, undo the preparatory step by re-introducing the original return statements, removing the added temp, break statements, and labeled block.
  - *Take care when re-introducing return statements in the (now void) modifier method, to avoid re-introducing the returned value.*
- Compile and test.
- Perform Inline Method on the refactored version of the selected method
  - *This will replace all calls to the original method with calls to the query and modifier. Note, however, that at each call site the query or modifier might be redundant, if the respective results are never used.*
  - *If the method is part of some inheritance relation (i.e. overriding a method in a superclass, being overriden in a subclass, or implementing a method declared in an interface), consider performing this refactoring throughout the hierarchy. Alternatively, consider skipping this step in such cases, leaving the calls to the extracted query and modifier in the original method.*
- Compile and test.

Fig. 14. Sliding-based mechanics for Separate Query from Modifier

co-slice shown on Fig. 3, is an example step toward SQfM. Successful completion of SQfM, in this case, would require a preliminary SQfM step to split the method `putIfAbsent()`, called on lines 19 and 22. This way, the slice would include a call to the query, contributing to the computation of `index`, whereas the co-slice would call the modifier, for effecting the fields of `charArrayCache` as needed.

5 Evaluation

To evaluate the potential of sliding for supporting refactorings, as presented above, in terms of the number of successful cases, the ability to compensate when the need arises, and the levels of code duplication, a preliminary experimentation to examine 55 cases has been performed, manually, on real Java code. The subject project has been the Java compiler in Eclipse. The comprehensive test suite of this compiler, featuring nearly 70,000 automated tests, with over 40,000 regression test cases, may provide us with some confidence regarding the correctness of the transformation steps.

As candidate sliding criteria, two kinds of slices were considered for extraction: slices with at least one partial loop, and slices of the value returned from a non-pure function. Isolation of the former exercises loop untangling through the Replace Temp with Query refactoring, while the extraction of the latter provides separation of commands from queries, exercising the Separate Query from Modifier refactoring.

The candidate criteria were identified as follows. For RTwQ, the Extract Method refactoring tool in Eclipse was employed on each loop, looking for rejected cases due to "ambiguous result". Such an error message is issued by the tool whenever more than one local variable is updated in the selected fragment (i.e. in the loop) and is live on exit from that fragment. A selection of 8 loops was examined. Those loops were found in 7 different methods of 6 different classes, with 2 local results in 4 cases and 3 results in the other 4 cases. The sliding and subsequent steps of RTwQ were performed on each local result, in sequence, with the fragment S being the full scope of the variable's declaration. When performing the extraction in a different order presented an important difference in the results, the alternative order was investigated too. In total, 23 cases of sliding for RTwQ were recorded this way.

For SQfM, any non-void method with side effects is a potential candidate, and 32 such cases, found in a package named `org.eclipse.jdt.internal.compiler.codegen` were examined.[1] All examples in this paper were taken from this experiment.

The resulting code, after the sliding step and the complete refactoring, in the successful cases, was tested and passed all tests. For sanity checking, deliberate mistakes were added to see that the code is indeed tested. Only one of the 38 succssfully refactored methods was not exercised by any test.

[1] Thanks to Alex Libov for automating the identification of both RTwQ and SQfM candidates, using Eclipse's JDT, its refactoring API, and the side effect analysis called *ModRef* in WALA.

In a preliminary step, the source code of many of the subject methods needed to be updated, to remove side effects from expressions, such as assignments within the predicates of `if` statements. The manual step would isolate the assignment into its own statement. This was not done exhaustively, but rather on demand, when the slice involved only the result of the expression or only the side effect. A future sliding algorithm and tool should better treat such cases correctly, without the need for manual update. Another type of manual change was to replace an early return statement with a break statement from a labeled block, as explained in Sect. 4 above. The change was later undone, after extraction of the slice and co-slice into new methods. One other type of manual change was to employ the SQfM refactoring on a called method in cases where the slice required only the result or only the side effect. And one final type of change was to move the declaration of a local variable, or to break two declarations in one statement into two separate declarations. In total, for 54 cases of sliding, a total of 77 manual changes were performed. (The total is 54, not 55 cases, due to the success in only 31 of the 32 SQfM cases, as will be explained below, added to the 23 RTwQ cases.)

The size of the code fragments ranged from 6 to 97 (in a method `numberOfDifferentLocals()` of class `StackMapFrame`), with an average of 23.5 statements per case. The slice size ranged from 1 statement (a return statement, returning a constant) to 85 (in the 97-statement method), averaging 10.4 statements. The co-slice size ranged from 1 to 82 (again in the same largest method) with an average of 20 statements, leading to an average of 6.9 duplicated statements per sliding.

In terms of the need for compensation, over 44% (7 RTwQ cases and 17 cases of SQfM) required no compensation at all. For all the remaining cases, variable localization and the introduction of backup variables (as discussed in Sect. 3.4 above) of at most 3 locals or fields per sliding case proved sufficient.

In terms of reuse, 132 out of the 239 total uses of an extracted variable were uses of a final value, averaging some 2.4 final uses and 1.99 non-final uses per case. After sliding, 46 non-final uses were left in the co-slices, leaving us with some 0.85 non-final uses per case.

The single most problematic case for SQfM, in class `ConstantPool`, involves the following code:

```
public byte[] dumpBytes() {
    System.arraycopy(this.poolContent, 0, (this.poolContent =
        new byte[this.currentOffset]), 0, this.currentOffset);
    return this.poolContent;
}
```

Sliding of the result without reuse of the allocated field array would fail, in this case, returning a reference to a different object than the field (with equal value). Another problem is that localization of the field array would require copying its initial value. A different type of separation is required here, possibly to compute the new field first, and only then return the result stored in that field.

6 Related Work

Earlier research on extracting slices from existing systems, in the context of software reverse engineering and reengineering, has focused mainly on how to discover reasonable slicing criteria [3,13]. In the context of refactoring tools, it is common to leave the choice of what to extract to the programmer.

The earliest mention of an interactive process for behavior-preserving method extraction [15,8] considered the extraction of contiguous code only.

Maruyama [14] proposed a scenario for the extraction of the slice of a single variable from a selected fragment, or block of statements, into a new method; a call to that method is placed ahead of the code for the remaining computation. The reuse of the extracted result was not of the final value only, but of any value defined by that variable. This way, the co-slice may make a reference intended for a non-final value, but get to use, instead, the final extracted value, making the transformation incorrect. This incorrectness was reported by Tsantalis et al. [17]; their more recent work constructs the complementary code in the same way, but defines PDG-based rules to identify these problematic cases and reject the transformation.

A number of provably correct algorithms for the extraction of a set of not-necessarily contiguous statements have been proposed in the literature [12,11,10].

Of those, tucking [12] is most generally applicable for isolating the slice of a code fragment. Tucking starts by adding to the statements designated for extraction all other statements in their slice, limited to a fragment that encloses those statements. If we apply this algorithm by selecting such a slice in the first place, no other statement would be added to the extracted code. This is unfortunately not the case in the algorithm of Komondoor and Horwitz [10], where each statement that the algorithm is unable to move away from the slice, correctly, is added to the extracted code. In the worst case, this approach extracts the whole fragment, essentially leaving it unchanged. In particular, no assignment can be duplicated and loop statements can either be extracted fully, or not extracted at all. Therefore, splitting loops as in our example of `def` and `pot`, is not possible by that algorithm. Komondoor and Horwitz had an earlier algorithm [11] in which all permutations of the selected statements were considered, in looking for an arrangement of statements in which all selected statements are contiguous and where all control and data dependences are preserved. This algorithm does not permit any duplication, not even of conditionals, and may therefore be applicable for slice extraction only in cases where each predicate in the slice appears in it along with all the statements it controls. So tucking is the only previous solution to slice extraction that can untangle a loop that computes more than one result, as in the RTwQ example of `def` and `pot` [5]. In tucking, however, the complementary code is computed as the slice from all non-extracted statements, so no reuse of the extracted results is possible. In our example of extracting the slice of `index` (see Fig. 1) from the full fragment of 1-23, that complement would include the whole fragment, as statement 23 would be included in the co-slice and then cause all the slice to be duplicated.

The idea of allowing data to flow from the extracted code to the complement, in sliding, is based on the two Komondoor and Horwitz algorithms [11,10].

7 Conclusion

To paraphrase Weiser's seminal work [19], sliding is a new way of recomposing programs automatically. Limited to code already written, it may prove useful during the refactoring, testing, and maintenance portions of the software life cycle. This paper concentrated on the basic methods for sliding programs and their embodiment in automatic tools for refactoring. Future work on sliding-based programming aids is necessary before the implications of this kind of recomposition are fully known.

Acknowledgements. I wish to thank Aharon Abadi, Yishai Feldman, Michiaki Tatsubori, and Shmuel Tyszberowicz for intriguing discussion and comments, and the anonymous reviewers for their helpful suggestions, all directing me at improvements to the paper.

My special thanks go to Cindy Eisner, Steve Fink, and Maayan Goldstein for the encouragement and crucial advice at key decision points.

References

1. An online refactoring catalog, http://www.refactoring.com/catalog/
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1988)
3. Cimitile, A., Lucia, A.D., Munro, M.: Identifying reusable functions using specification driven program slicing: a case study. In: ICSM, pp. 124–133 (1995)
4. Ettinger, R.: Refactoring via Program Slicing and Sliding. Ph.D. thesis, University of Oxford, Oxford, United Kingdom (2006)
5. Ettinger, R., Verbaere, M.: Untangling: a slice extraction refactoring. In: AOSD 2004: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, pp. 93–101. ACM Press, New York (2004)
6. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987), http://doi.acm.org/10.1145/24039.24041
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (2000)
8. Griswold, W., Notkin, D.: Automated assistance for program restructuring. ACM Transactions on Software Engineering 2(3), 228–269 (1993)
9. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12(1), 26–60 (1990)
10. Komondoor, R., Horwitz, S.: Effective automatic procedure extraction. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension (2003)
11. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: POPL 2000: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 155–169. ACM Press, New York (2000)

12. Lakhotia, A., Deprez, J.C.: Restructuring programs by tucking statements into functions. Information and Software Technology 40(11-12), 677–690 (1998), citeseer.nj.nec.com/lakhotia99restructuring.html
13. Lanubile, F., Visaggio, G.: Extracting reusable functions by flow graph-based program slicing. IEEE Trans. Software Eng. 23(4), 246–259 (1997)
14. Maruyama, K.: Automated method-extraction refactoring by using block-based slicing, pp. 31–40. ACM Press (2001)
15. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, IL, USA (1992), citeseer.nj.nec.com/opdyke92refactoring.html
16. Ottenstein, K., Ottenstein, L.: The program dependence graph in a software development environment. In: Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 177–184 (1984)
17. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities. In: CSMR 2009: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, pp. 119–128. IEEE Computer Society, Washington, DC (2009)
18. Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a scripting language for refactoring. In: ICSE, pp. 172–181 (2006)
19. Weiser, M.: Program slicing. In: ICSE, pp. 439–449 (1981)

Static Detection of Loop-Invariant Data Structures

Guoqing Xu[1], Dacong Yan[2], and Atanas Rountev[2]

[1] University of California, Irvine, CA, USA
[2] Ohio State University, Columbus, OH, USA

Abstract. As a culture, object-orientation encourages programmers to create objects, both short- and long-lived, without concern for cost. Excessive object creation and initialization can cause severe runtime bloat, which degrades significantly application performance and scalability. A frequently-occurring coding pattern that may lead to large volumes of (temporary) objects is the creation of objects that, while allocated per loop iteration, contain values independent of specific iterations. Finding these objects and moving them out of loops requires sophisticated interprocedural analysis, a task that is difficult for traditional dataflow analyses such as loop-invariant code motion to accomplish.

Our work targets *data structures* that are loop-invariant, and presents a static type and effect system to detect loop-invariant data structures. For each loop, our analysis inspects each logical data structure in order to find those that have disjoint instances per loop iteration and contain loop-invariant data. Instead of automatically hoisting them to improve performance (which is over-conservative), we report *hoistability measurements* for each disjoint loop data structure detected by our analysis. Eventually these data structures are ranked based on these measurements and are presented to the user to help manual tuning. We have performed a variety of studies on a set of 19 moderate/large-sized Java benchmarks. With the help of hoistability measurements, we found optimization opportunities in most of the programs that we inspected and achieved significant performance improvements in some of them (e.g., 82.1% running time reduction).

1 Introduction

As a culture of object-orientation, Java programmers are taught to freely create objects for whatever tasks they want to achieve, without concern for cost. They often take for granted that the runtime system can optimize away all execution inefficiencies: the Just-In-Time (JIT) compiler can remove whatever redundancy exists in the code, and the Garbage Collector (GC) can quickly reclaim redundant objects created for simple tasks. However, creating an object in Java with a `new` operator, in most cases, is far beyond allocating memory space, and can be much more expensive than a programmer realizes.

For example, object creation may need to execute large volumes of code to construct and initialize a data structure, and this process may even involve many slow I/O operations. One especially important case is when these expensive objects have data that is invariant. Frequently constructing data structures with unchanged data may have significant effect on application running time and scalability. Large improvements can often be seen when these data structures are reused rather than recreated.

Loops are places where such data structures can cause significant harm and thus special attention needs to be paid to find and optimize them. We propose static analyses

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 738–763, 2012.

```
for(int i = 0; i < N; i++){
      SimpleDateFormat sdf = new SimpleDateFormat();
      try{
            Date d = sdf.parse(date[i]);
            ...
        }catch(...) {...}
}
                                        (a)

Templates _template = factory.newTemplates(stylesheet);
while(...){
      XMLFile file = getNewInputFile();
      XMLTransformer transformer = _template.newTransformer();
      transformer.transform(file);
      ...
}
                                      (b)
```

Fig. 1. Real-world examples of heavy-weight creation of loop-independent data structures. (a) A `SimpleDateFormat` object is created inside the loop to parse the given `Date` objects; (b) An `XMLTransformer` object is created within the loop to transform the input XML files.

that can find data structures that are created in a loop but are independent of specific iterations. This work is motivated by bloat patterns that are regularly seen in large-scale applications. Figure 1 shows two examples extracted from the real-world programs that we have studied. The code pattern in part (a) has appeared a great number of times in applications that were written by IBM's customers and tuned by a group from IBM Research [1]. The programmer may have never realized that creating one `SimpleDateFormat` object requires to load many resource bundles to get the current date, compile the default date pattern string, and load the time zone to create a calendar. The process involves many expensive operations such as object clones, hash table lookups, etc. Part (b) illustrates a problem detected by our tool in `DaCapo/xalan`. An `XMLTransformer` object is created in a loop to transform the input XML file. While the input file is updated per loop iteration, the transformer object is loop-invariant. A great amount of effort is needed to create a transformer and significant performance improvement can be achieved after hoisting the creation of this transformer. Details of this example can be found in Section 5.

Technical Challenges. While loop optimizations have been extensively studied and used in modern optimizing compilers [2], they are mostly intraprocedural and deal only with instructions that operate on scalar variables and simple data structures (e.g., arrays and linked lists). They are far from reaching our goal of finding large optimization opportunities in programs that make extensive use of object-oriented data structures. Techniques such as loop-invariant code motion target instructions whose input variables are not defined in the loop. Such techniques are usually ineffective at handling instructions involving objects: for an object created in the loop, even though one of its fields used in an instruction is not defined, it is not safe to move this instruction out of the loop, as other fields of the object may be modified elsewhere in the loop. In an object-oriented program, data abstractions are much more complex and data in different locations are tightly coupled based on logical object models.

Focusing on Logical Data Structures. In this work, we focus on the data side of the hoisting problem, that is, to find logical data structures that are loop-invariant, regardless

of whether or not it is possible to hoist the actual code statements that access these data structures. If a logical data structure is loop-invariant, the programmer should modify its creating and accessing code statements in order to move it out of the loop. There are two important aspects in determining whether a logical data structure is hoistable. First, it is critical to understand *how this data structure is built up*. For example, all objects in a hoistable data structure have to be allocated together in one iteration of the loop. In addition, any object in a hoistable data structure must be owned only by this data structure, and it cannot escape to other data structures. These properties can be verified by checking *points-to relationships* among objects. Second, it is important to understand *where this data structure gets its values from*. For example, all values contained in (heap locations of) a hoistable data structure must *not* be computed from any loop-iteration-specific value. This aspect of the problem is naturally related to the *data dependence* problem, and thus, such (value origin) properties can be verified by checking *data-dependence* relationships.

These two kinds of relationships are formalized as two (points-to and dependence) effects by a type and effect system presented in Section 3. As the identification of loop-invariant data structures requires to reason about whether objects connected by these relationships are always created in the same iteration of a loop, our analysis computes, for each loop object, a *loop iteration count abstraction* that indicates whether or not an instance of the object created in one iteration of the loop can be carried over to the next iteration. A more detailed description of this abstraction can be found in Section 2. Section 3 presents a formalism that computes such abstractions.

Manual Tuning with the Help of Hoistability Measurement. Given logical loop-invariant data structures identified by our analysis, the second challenge lies in how to perform the actual hoisting. While it is attractive to design a transformation technique that automatically pulls out invariant data structures, we found that there is little hope that a completely automated approach can effectively hoist these data structures in practice. This is first because of the over-conservative nature of any transformation technique, which may prevent the technique from hoisting many real-world loop-invariant data structures due to their complex usage in large-scale applications. The chance of developing an effective transformation technique becomes even smaller in the presence of the many Java dynamic features such as dynamic class loading and reflection. Second, effectively optimizing real-world data structures requires developer insight. For example, a data structure with 100 fields cannot be transformed if it has even a single non-loop-invariant field. In fact, by manually inspecting and perhaps modifying the data model, it is highly likely that the data structure can be made hoistable (i.e., by introducing a separate object to store that loop-dependent field).

Our work advocates a semi-automated approach that is intended to identify larger optimization opportunities by bringing developer insight into the optimization process. Instead of eagerly looking only for *completely-hoistable* logical data structures, we also identify *partially-hoistable* logical data structures, by computing a *hoistability measurement* for each logical data structure, and rank all such data structures based on these measurements to help manual tuning. The higher measurement a data structure has, the more likely it is that this data structure can be manually hoisted.

One additional advantage of manual tuning using hoistability measurements is that these metrics can be easily modified to incorporate dynamic information obtained from a profile. Section 4 presents one such modification that includes loop frequencies in the metrics so that the "loop hotness" factor is taken into account when the rank of a data structure is computed. To optimize a non-hoistable data structure, the programmer can either split the data model (e.g., to separate the loop-invariant fields and non-invariant fields) and/or restructure the statements that access it (e.g., to eliminate dependences between hoistable and non-hoistable statements). In addition, highly-ranked data structures can often be indicators of other loop-related inefficiencies, such as inappropriate implementation choices. These problems may also be revealed during the inspection of the reported data structures.

Evaluation. We evaluated our technique using a set of 19 Java programs. With the help of hoistability measurements, we found optimization opportunities in most of these programs. We discuss the performance gains we have achieved for five representative programs: `ps`, `xalan`, `bloat`, `soot-c`, and `sablecc-j`. For example, we found a performance problem in `DaCapo/xalan`; removing it can improve the benchmark performance by 10.1%. As another example, we found a bottleneck in the core components of `ps`. After the optimization, the running time was reduced by 82.1%. Detailed description of the empirical evaluation can be found in Section 5. These results indicate that the proposed technique can be useful both in the coding phase (for finding small performance issues before they pile up) and in the tuning phase (for identifying performance bottlenecks).

2 Overview

Figure 2 shows a simple running example. This example contains 10 allocation sites (including string literals), and all of them are located in loops (i.e., either directly in a loop or in a method invoked in a loop). Our analysis inspects each object[1] located in a loop and discovers its structure (i.e., including objects that are calling-context-sensitively reachable from it) using context-free language (CFL)- reachability. Using new CFL-reachability algorithms, we develop novel techniques that inspect individual objects in a loop, identify their structures while taking into account the calling contexts of methods, and compute their hoistability, all without requiring a pre-computed whole-program points-to solution.

Points-to Relationships. Figure 3(a) illustrates three data structures (a.1), (a.2), and (a.3) that are rooted at objects in the two loops shown in Figure 2. Each object is given a name o_i, where i is the number of the line in the code where the object is created. Each edge in a data structure represents a points-to relationship, and is annotated with a field name and a pair of integers (i, j). Field `elm` is a special field used to represent array elements. Integers in this pair are the loop *iteration count abstractions* (ICAs) for the two objects connected by this edge, and they can be used to determine whether these objects are created in the same iteration of the loop. Following the iteration abstraction [3] and the recency abstraction [4], an ICA can be one of three (abstract) values: 0, 1, or $\top$. Note

[1] "Object" will be used in the rest of the paper to denote a static abstraction (i.e., an allocation site), while "instance" denotes a run-time object.

```
1 class List{
2   Object[] arr; int index = 0;
3   List() { arr = new Object[1000];}
4   void add(Object o){ if(index < 1000) arr[index++] = o;}
5   Object get(i){ return arr[i]; }
6 }
7 class Pair{
8   Object f; Object g;
9   Pair(Object o1, Object o2){ this.f = o1; this.g = o2;}
10}
11
12 class Client{
13   static void main(String[] args){
14     for(int i = 0; i < 500; i++){
15       List l = new List();
16       Pair p = new Pair("hello", "world");
17       l.add(p);
18     }
19     Integer b = null;
20     for(int j = 0; j < 400; j++){
21       Integer a = new Integer(j);
22       if(j == 20) b = new Integer(10);
23       Pair q = new Pair(a, b);
24       Pair r = new Pair("good", a);
25       ... //use q and r
26     }
27  }
```

Fig. 2. Running example

that the use of ICA is not a contribution of this paper. The three major contributions are (1) a novel calling-context-sensitive algorithm for computing points-to and dependence relationships that are annotated with ICA information, (2) a new technique for detecting loop-invariant data structures with the help of such points-to/dependence information, and (3) the development of quantitative measurements that use these relationships to help programmers identify hoistable data structures.

For a particular loop l, an object whose ICA is 0 with respect to l must be created outside l. The ICA for an object being either 1 or $\top$ (with respect to l) means the object must be created inside l. In particular, let us consider a run-time iteration p of l and a run-time instance r created by allocation site o_r such that r is live during the execution of p. If the ICA for o_r is 1, r is guaranteed to be created during iteration p. In other words, the ICA for an object being 1 indicates that its instances must be "fresh" across iterations, that is, in any iteration where an instance of it is live, this instance must be created in that iteration (i.e., it must *not* be carried over from a previous iteration). An object that has a $\top$ in its ICA is created in a (previous) unknown iteration. For example, the ICA for o_{23} is 1, as it creates a fresh object in each iteration of the loop at line 20. The ICA for o_{22} is $\top$, as the instance it creates in one iteration can be carried over to the next iteration.

Hence, for a points-to edge annotated with (i, j), $i = j = 1$ guarantees that the two objects connected by the edge must be created in the same iteration of the loop, while either i or j being $\top$ indicates that the two objects may be created in different iterations. A data structure is obviously not hoistable if it contains objects that are created in different iterations.

Dependence Relationships. Figure 3(b) illustrates the dependence (def-use) chains that start at memory locations in each data structure shown in Figure 3(a). An edge $o_1.f \leftarrow o_2.g$ indicates that a run-time value contained in a heap location abstracted by

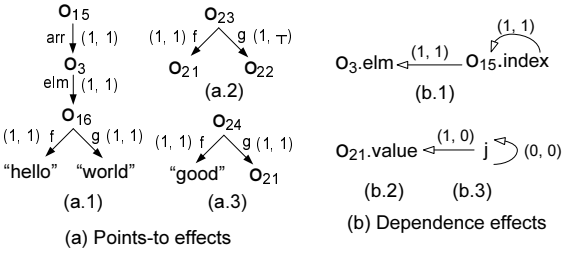


Fig. 3. Data structures identified for the running example and their effect annotations. (a) Points-to effects among objects; (b) Dependence effects among memory locations.

$o_2.g$ is required in computing a value written into (a heap location abstracted by) $o_1.f$. Note that o_{21}.*value* is a field of class `Integer` that stores the int value embedded in an `Integer` object. A stack location is also considered in a dependence chain, if this location (variable) has a primitive type and is in the method that contains the loop of interest (e.g., variable j in method `main`). Such a variable needs to be taken into account as it may contain iteration-specific values. Variables that are not in the loop-containing method are abstracted away from dependence chains, as they can get iteration-specific values only from heap locations or variables in the loop-containing method (e.g., variable o at line 4). Reference-typed variables (e.g., a and b) do not need to be considered as well (even though they are in the loop-containing method), because they can get loop-specific values only from heap locations, and thus, it is necessary to track only heap locations.

Note that each dependence edge $o_1.f \leftarrow o_2.g$ is also annotated with a pair of ICAs (for o_1 and o_2), which is used to determine whether o_1 and o_2 are always created in the same loop iteration. If a node in a dependence edge is a stack variable, such as j, its ICA is determined by whether or not it is declared in the loop. For example, j's ICA is 0, because it is declared before the loop starts. Its ICA would have been 1 if it were declared in the loop. The ICA for a stack variable can never be $\top$, as each variable declared in a loop must be initialized (i.e., get a new value) per iteration.

Hoistable Logical Data Structures. As discussed earlier in Section 1, the analysis identifies (completely or partially) hoistable logical data structures from a *purely data perspective*, regardless of the actual code and control flow. This can be done by reasoning about these two kinds of annotated relationships. A hoistable data structure has the following important properties.

(1) *(Disjoint)*. Its run-time instances, created by different iterations of the loop, have to be disjoint. No object is allowed to appear in multiple instances of one single logical data structure. This property can be verified by checking whether all points-to edges in a data structure are annotated with $(1, 1)$. A data structure is not hoistable if any of its nodes has a non-1 ICA. This guarantees that any instance of a hoistable data structure does not have objects created outside the loop or in different iterations. For example, (a.2) is not hoistable, as edge $o_{23} \xrightarrow{g} o_{22}$ may connect objects created in different iterations.

(2) *(Loop-invariant)*. Fields of objects in a hoistable data structure have to be loop-invariant. No data in any run-time instance of the data structure can be dependent on specific loop iterations. We check this property by formulating it as a data-dependence

problem. A sufficient condition for the statement "object o is loop-invariant" is that, for each field of o, (2.1) no edge on a dependence chain (e.g., shown in Figure 3(b)) that starts from the field can have $\top$ in its annotated ICA pair, and (2.2) for each memory location node $o.f$ (i.e., heap location) or j (i.e., stack location) on the chain, if the ICA for o or j is 0, this node must not be involved in a *dependence cycle*.

(2.1) enforces that all (stack and heap) locations from which a hoistable data structure instance (allocated in one iteration) gets its data are either created in this same iteration, or exist before the loop starts. No data in this instance can be obtained from an object created in a different iteration. In addition, as stated in (2.2), if one such location already exists before the loop starts (e.g., variable j), this node must not be in a dependence cycle. Otherwise, its value may be updated by each iteration and any data structure that is dependent on this value is not hoistable. For example, in Figure 3(b), o_{21}.*value* depends on variable j, whose ICA is 0. Because j is in a dependence cycle, o_{21}.*value* may have iteration-specific values, and thus, any data structure that contains o_{21} is not hoistable (e.g., structures (a.2) and (a.3) in Figure 3(a)). Note that field o_3.*elm* is loop-invariant: while it depends on field o_{15}.*index*, which is involved in a cycle, o_{15}'s ICA is 1. Hence, it is impossible for an iteration-specific value to propagate to this field.

Note that these two properties are sufficient (but not necessary) conditions for hoistable logical data structures. For example, the first condition (i.e., disjointness) is an over-conservative approximation of the shape of a hoistable data structure—it is perfectly possible for a hoistable data structure to contain objects that are created outside the loop (i.e., their ICAs are 0) but not mutated in the loop. We choose not to consider such objects in our hoistable data structure definition primarily for scalability purposes—these objects (created outside the loop) may have long dependence chains (as objects that they reference can come from arbitrary places). On the contrary, dependence chains for objects that are created in the loop and do not escape the loop (i.e., all their ICAs are 1) are generally much shorter. Therefore, the dependence analysis is much more scalable when considering only these chains.

Computing Hoistability Measurements. After inspecting these two conditions, it is clear that only data structure (a.1) is a completely hoistable logical data structure. However, there might still exist optimization opportunities with the other two (partially hoistable) data structures. For example, if we can move object o_{22} out of data structure (a.2), it may still be possible to hoist (a.2). In order to help programmers discover such hidden optimization opportunities, hoistability measurements are proposed to quantify the likelihood of manually hoisting data structures out of loops.

For example, for each data structures shown in Figure 3, we compute two separate hoistability measurements based on the two orthogonal (points-to and dependence) effects mentioned above: structure-based hoistability (SH) that considers how many objects in the data structure must be allocated in the same loop iteration (i.e., that comply with condition 1), and dependence-based hoistability (DH) that considers how many fields in the data structure must contain loop-invariant data (i.e., that comply with condition 2). Eventually, these three data structures are ranked based on the two measurements and are then presented to the user for further inspection. Detailed description of hoistability measurements can be found in Section 4.

$$
\begin{array}{lll}
\text{Variables} & a, b \in & \mathsf{V} \\
\text{Allocation sites} & o \in & \mathsf{O} \\
\text{Instance fields} & f \in & \mathsf{F} \\
\text{Labels} & l \in & \mathsf{L} \\
\text{Statements} & e \in & \mathsf{E}
\end{array}
$$

$$
\begin{array}{l}
e ::= \; a = b \mid a = new\ ref^{o} \mid a = b.f \mid a.f = b \mid a = \mathsf{null} \mid \\
\qquad e\ ;\ e \mid \text{if (*) then } e \text{ else } e \mid \text{while}^{l} \text{ (*) do } e
\end{array}
$$

(a)

$$
\begin{array}{llll}
\text{Iteration count} & i & ::= & 0 \mid 1 \mid 2 \mid \ldots \quad \in \mathsf{N} \\
\text{Iteration map} & \nu & \in & \mathsf{L} \to \mathsf{N} \\
\text{Loop status} & \pi & ::= & \langle l\ , i \rangle \qquad l \in \mathsf{L} \cup \{0\} \\
\text{Labeled object} & \hat{o} & ::= & o^{\pi} \qquad \in \Phi \\
\text{Heap} & \sigma & \in & \Phi \times \mathsf{F} \to \Phi \cup \{\bot\} \\
\text{Environment} & \rho & \in & \mathsf{V} \to \Phi \cup \{\bot\} \\
\text{Data origin} & \mu & \in & \mathsf{V} \to 2^{\Phi \times \mathsf{F}} \\
\text{Heap points-to effect} & H & ::= & \emptyset \mid H \cup \{\hat{o}_1 \rhd^{f} \hat{o}_2\} \\
\text{Heap data dep. effect} & \Omega & ::= & \emptyset \mid \Omega \cup \{\hat{o}_1.f \prec \hat{o}_2.g\}
\end{array}
$$

(b)

Fig. 4. A simple `while` language: (a) abstract syntax; (b) semantic domains

3 Loop-Invariant Logical Data Structures

This section formalizes the notion of loop-invariant logical data structure, and in this context, formally defines our analysis that identifies hoistable data structures. The presentation proceeds in three steps. First, we define a simple imperative language and present its abstract syntax and operational semantics, which we will use to formalize our analysis algorithms. For the ease of presentation, function calls are not considered in this language. Our implementation supports full context-sensitivity using a CFL-reachability formulation.

Second, we present a type and effect system that abstracts concrete objects and effects. Finally, the analysis that detects hoistable data structures is described based on the abstract heap effects generated by the type and effect system.

3.1 Language, Semantics, and Effect System

Language. The abstract syntax and the semantic domains for the simple `while` language that we use are defined in Figure 4. A program in this language has a fixed set of global variables with reference types. While primitive-typed variables are considered in our analyses, they are excluded from this language for the simplicity of presentation. Each allocation site is labeled with an ID o. Each loop is annotated with a natural number label l ($l > 0$), which will be used as the ID of the loop. * denotes a side-effect-free boolean expression that contains only local variables and constants.

We develop a concrete operational semantics for the language in order to detect hoistable data structures. A loop iteration count i records the number of iterations that a loop has executed. A global loop iteration map ν maps each loop (label) to its current iteration count. Each object instance is represented as its allocation site o annotated

with a pair $\langle l, i\rangle$, where l is the label of the loop in which o is located (always > 0), and i is the count of the iteration of l that creates this instance. If an object is not located in any loop, the loop status π for its instances is always $\langle 0, 0\rangle$. Our analysis does not consider hoisting objects out of nested loops, and in the presentation we assume that all loops in the abstract language are not nested. While nested loops can be handled easily in our framework (e.g., by creating and associating with each object an *iteration count map* that records an iteration count for each loop in which the object is located), we found that it is not useful in hoisting data structures for real-world Java programs: it is extremely rare that a data structure can be hoisted out of multiple loops.

A heap σ records object reference relationships, and an environment ρ maps variables to objects in the heap. They are defined in standard ways. A data origin map μ records, for each stack variable v, a set of *heap locations* such that values in these locations are required (i.e., either as a pointer for dereferencing, or being copied through a sequence of intermediate stack locations) in order to obtain a value written to v. This map tracks dependences between variables and their relevant heap locations, and will be used to compute *dependence effects* as described shortly. For example, after the execution of a sequence of statements $c = d.f; b = c; a = b$, we have $\mu(a) = \mu(b) = \mu(c) = \{o_d.f\} \cup \mu(d)$, where o_d represents the object that d points to. $\mu(d)$ is included here because d is required as a reference to an object from which the value is obtained. Dependences via the intermediate stack locations (e.g., b and c) are abstracted away as we are interested only in fields of objects that form data structures.

Note that μ records only one-hop heap location dependence—if the value in $d.f$ is obtained from another heap location, $\mu(a)$, $\mu(b)$ and $\mu(c)$ remain the same. Multi-hop heap location dependences can be obtained by computing the transitive closure of μ.

As discussed earlier in Section 2, we use *points-to* and *dependence* relationships to reason about (1) how data structures are built up and (2) where they get values from, respectively. These relationships are modeled by the following two kinds of effects in our system. A heap *points-to effect* $\hat{o}_1 \rhd^f \hat{o}_2 \in H$ is generated if, at a certain point, object $\hat{o}_2$ becomes reachable from $\hat{o}_1$ through field f. A *data dependence effect* tracks the flow of data. One such effect $\hat{o}_1.f \prec \hat{o}_2.g \in \Omega$ indicates that $\hat{o_2}.g$ is required in order to compute a value written into $\hat{o_1}.f$. This effect captures a transitive data dependence relationship between two heap locations, abstracting away a possible sequence of dependences via intermediate stack variables. Data dependence effects can be computed efficiently by using the data origin map μ.

Note that in this language, it is safe for a dependence chain to *not* include any stack variable (like j in Figure 3 (b)). This is because the language supports only reference-typed variables, which can never form a dependence cycle themselves (without a heap location involved). While we choose not to include primitive types in this language (for the simplicity of presentation), our implementation handles both primitive and reference types. It is also important to note that the effects shown in Figure 4 are *concrete effects*. We will present an approach to project them into *abstract effects*, which are essentially the annotated edges shown in Figure 1.

Concrete Instrumented Semantics. Figure 5 presents a big-step operational semantics for our language. A judgment of the form

$$e, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega$$

$$a = \text{null}, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \bot], \mu[a \mapsto \emptyset], \emptyset, \emptyset \quad (\textsc{Assign-Null})$$

$$\frac{\rho' = \rho[a \mapsto \hat{o}] \qquad \sigma' = \sigma[\forall f.(\hat{o}.f \mapsto \bot)] \qquad \hat{o}.o = alloc \qquad \hat{o}.\pi = \langle l\ ,\ \nu(l)\rangle}{a = \text{new ref}^{alloc}, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma', \rho', \mu[a \mapsto \emptyset], \emptyset, \emptyset} \quad (\textsc{New})$$

$$a = b, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \rho(b)], \mu[a \mapsto \mu(b)], \emptyset, \emptyset \quad (\textsc{Assign})$$

$$\frac{\rho(b) = \hat{o} \qquad \mu' = \mu[a \mapsto \mu(b) \cup \{\hat{o}.f\}]}{a = b.f, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \sigma(\hat{o}.f)], \mu', \emptyset, \emptyset} \quad (\textsc{Load})$$

$$\frac{\begin{array}{c}\rho(a) = \hat{o}_1 \qquad \rho(b) = \hat{o}_2 \\ H = (\hat{o}_2 = \text{null}\ ?\ \emptyset : \{\hat{o}_1 \rhd^f \hat{o}_2\}) \qquad \Omega = \bigcup\{\hat{o_1}.f \prec \hat{o_i}.g_i \mid \hat{o}_i.g_i \in \mu(a) \cup \mu(b)\}\end{array}}{a.f = b, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma[\hat{o}_1.f \mapsto \hat{o}_2], \rho, \mu, H, \Omega} \quad (\textsc{Store})$$

$$\frac{e_1, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H_1, \Omega_1 \qquad e_2, \nu', \sigma', \rho', \mu' \Downarrow \nu'', \sigma'', \rho'', \mu'', H_2, \Omega_2}{e_1; e_2, \nu, \sigma, \rho, \mu \Downarrow \nu'', \sigma'', \rho'', \mu'', H_1 \cup H_2, \Omega_1 \cup \Omega_2} \quad (\textsc{Comp})$$

$$\frac{e_1, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega}{\text{if } (*) \text{ then } e_1 \text{ else } e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega} \quad (\textsc{If-Else-1})$$

$$\frac{e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega}{\text{if } (*) \text{ then } e_1 \text{ else } e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega} \quad (\textsc{If-Else-2})$$

$$\frac{\begin{array}{c}e, \nu[j \mapsto \nu(j)+1], \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H_1, \Omega_1 \\ \text{while}^j\ (*) \text{ do } e, \nu', \sigma', \rho', \mu' \Downarrow \nu'', \sigma'', \rho'', \mu'', H_2, \Omega_2\end{array}}{\text{while}^j\ (*) \text{ do } e, \nu, \sigma, \rho, \mu \Downarrow \nu'', \sigma'', \rho'', \mu'', H_1 \cup H_2, \Omega_1 \cup \Omega_2} \quad (\text{W})$$

Fig. 5. Concrete instrumented semantics

starts with a statement e, which is followed by loop iteration map ν, heap σ, environment ρ, and value origin map μ. The execution of e terminates with a final iteration map ν', heap σ', environment ρ', origin map μ', heap points-to effect set H, and heap data dependence effect set Ω.

Rules COMP, IF-ELSE1, IF-ELSE2, and W are defined in expected ways. In rule ASSIGN-NULL and NEW, the data origin for a (in μ) is assigned $\emptyset$, as the value is freshly generated and does not depend on any heap value. In rule NEW, for a labeled object $\hat{o}$, $\hat{o}.o$ and $\hat{o}.\pi$ denote the allocation site and the loop status pair for $\hat{o}$, respectively. In $\hat{o}.\pi$, the loop l where the allocation site o is located is determined statically, and is associated with each instance created by o. The iteration count for l is retrieved from the global iteration count map ν. Rule ASSIGN propagates both the object reference and the data origin of this reference value from b to a. In rule LOAD, the data origin map μ for variable a is updated in a way so that both $\hat{o}.f$ and the origin of the value in b are recorded as a's origin. Hence, dependences via both the value copy (from $b.f$ to a) and the pointer dereference (i.e., dereferencing b) are captured.

To handle a store $a.f = b$ where b's value is written to the heap, a points-to effect $\hat{o}_1 \rhd^f \hat{o}_2$ is first generated. The rule next generates a set of heap dependence effects $\{\hat{o}_1.f \prec \hat{o}_i.g_i \mid \hat{o}_i.g_i \in \mu(b) \cup \mu(a)\}$, by consulting the data origin map μ. Each dependence effect states that a value read from field g_i of object $\hat{o}_i$ has been used to produce a value written to $\hat{o}_1.f$ during the execution.

► **Example** For illustration, consider the following example:

```
a = new ref^o1; e = new ref^o2; a.f = e; j = a.f;
while^1 (j) do{ b = a.f; d = b; c = new ref^o3; c.g = d;}
```

Iteration count abstr.	$\tilde{i}$	$::= 0 \mid 1 \mid \top$	$\in \mathbb{N}$
Loop status abstr.	$\tilde{\pi}$	$::= \langle l, \tilde{i} \rangle$	$l \in \mathsf{L} \cup \{0\}$
Type	$\tilde{\tau}$	$::= o^{\tilde{\pi}} \mid \top$	$\in \mathbb{T}$
Type environment	Γ	$\in \mathsf{V} \to \mathbb{T} \cup \{\bot\}$	
Data origin abstr.	$\tilde{\mu}$	$\in \mathsf{V} \to 2^{\mathbb{T} \times \mathsf{F}}$	
P.T. effect abstr.	$\tilde{H}$	$::= \emptyset \mid \tilde{H} \cup \{\tilde{\tau_1} \trianglerighteq \tilde{\tau_2}\}$	
Dep. effect abstr.	$\tilde{\Omega}$	$::= \emptyset \mid \tilde{\Omega} \cup \{\tilde{\tau_1}.f \preceq \tilde{\tau_2}.g\}$	

Fig. 6. Syntax of types and abstract effects

At the end of the first iteration of the loop, the semantic domains contain the following values:

$\nu = [1 \mapsto 1]$,
$\sigma = [\hat{o_1}.f \mapsto \hat{o_2}, \hat{o_3}.g \mapsto \hat{o_2}]$,
$\rho = [a \mapsto \hat{o_1}, b \mapsto \hat{o_2}, c \mapsto \hat{o_3}, d \mapsto \hat{o_2}, e \mapsto \hat{o_2}, j \mapsto \hat{o_2}]$,
$\mu = [a \mapsto \emptyset, b \mapsto \{\hat{o_1}.f\}, c \mapsto \emptyset, d \mapsto \{\hat{o_1}.f\}, e \mapsto \emptyset, j \mapsto \{\hat{o_1}.f\}]$,
$H = \{\hat{o_1} \rhd^f \hat{o_2}, \hat{o_3} \rhd^g \hat{o_2}\}$,
$\Omega = \{\hat{o_3}.g \prec \hat{o_1}.f\}$. ◀

Abstract Semantics. The concrete semantics uses an unbounded number of objects and unbounded loop iteration counts. We next develop a type and effect system that describes an abstract semantics, which conservatively approximates the concrete semantics with a bounded set of objects and bounded loop iteration counts. The syntax of types and abstract effects are illustrated in Figure 6. The abstraction of each concrete domain (e.g., π) shown in Figure 4 is represented by its corresponding tilded symbol (e.g, $\tilde{\pi}$). Environment ρ is abstracted by the type environment, denoted by Γ. A type $\tilde{\tau}$ abstracts a labeled object instance $\hat{o}$ by projecting its concrete iteration count $\hat{o}.\pi.i$ to an iteration count abstraction (ICA) $\tilde{\tau}.\tilde{\pi}.\tilde{i}$, which can have three abstract values: 0, 1, and $\top$. The meaning of these values was explained in Section 2. Using this type and effect system, we can identify data structures whose objects are guaranteed to be created in the same iteration by reasoning about object ICAs. Note that each abstract effect in $\tilde{H}$ and in $\tilde{\Omega}$ corresponds to an edge in Figure 3 (a) and in Figure 3 (b), respectively.

Figure 7 shows the type rules, which are parallel with the operational semantics in Figure 5. Auxiliary operations used in the type rules are shown in Figure 8. Some abstract semantic domains in Figure 6 are extended with $\top$ and/or $\bot$ elements, as necessary.

Since the type and effect system abstracts the concrete semantics in Figure 5, most of the rules in Figure 7 are similar to their corresponding operational semantics rules. Here we explain only a few of them that differ significantly from their concrete counterparts. In rule TNEW, the ICA for a newly created object is always 1, and this value will be changed later (by rule TW) if this object is carried over to the next iteration. (For objects created outside of loops, the ICA is 0; for brevity, this variation of TNEW is not shown in Figure 7.) Rule TLOAD types variable a with an unknown type $\top$. This handling is over-conservative for the purpose of soundness. Our implementation improves this by consulting a points-to graph that is computed on demand.

Type environment join ($\uplus$) needs to be performed in order to handle different control flow paths of an `if-else` statement (in Rule TIF-ELSE). Joining two environments

$$\Gamma, \tilde{\mu} \vdash a = \text{null} : \Gamma'[a \mapsto \bot], \tilde{\mu}[a \mapsto \emptyset], \emptyset, \emptyset \quad \text{(TASSIGN-NULL)}$$

$$\frac{\Gamma' = \Gamma[a \mapsto \tilde{\tau}] \qquad \tilde{\tau}.o = alloc \qquad \tilde{\tau}.\tilde{\pi} = \langle l\ ,\ 1\rangle}{\Gamma, \tilde{\mu} \vdash a = \text{new ref}^{alloc} : \Gamma', \tilde{\mu}[a \mapsto \emptyset], \emptyset, \emptyset} \quad \text{(TNEW)}$$

$$\Gamma, \tilde{\mu} \vdash a = b : \Gamma[a \mapsto \Gamma(b)], \tilde{\mu}[a \mapsto \tilde{\mu}(b)], \emptyset, \emptyset \quad \text{(TASSIGN)}$$

$$\frac{\tilde{\tau} = \Gamma(b) \qquad \tilde{\mu}' = \tilde{\mu}[a \mapsto \tilde{\mu}(b) \cup \{\tilde{\tau}.f\}]}{\Gamma, \tilde{\mu} \vdash a = b.f : \Gamma[a \mapsto \top], \tilde{\mu}', \emptyset, \emptyset} \quad \text{(TLOAD)}$$

$$\frac{\begin{array}{c}\tilde{\tau}_1 = \Gamma(a) \qquad \tilde{\Omega} = \bigcup\{\tilde{\tau}_1.f \preceq \tilde{\tau}_i.g_i \mid \tilde{\tau}_i.g_i \in \tilde{\mu}(b) \cup \tilde{\mu}(a)\} \\ \tilde{\tau}_2 = \Gamma(b) \qquad \tilde{H} = \{\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2\} \text{ if } \tilde{\tau}_1 \neq \bot \text{ and } \tilde{\tau}_2 \neq \bot,\ \emptyset \text{ otherwise}\end{array}}{\Gamma, \tilde{\mu} \vdash a.f = b : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}} \quad \text{(TSTORE)}$$

$$\frac{\Gamma, \tilde{\mu} \vdash e_1 : \Gamma', \tilde{\mu}', \tilde{H}_1, \tilde{\Omega}_1 \qquad \Gamma', \tilde{\mu}' \vdash e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_2, \tilde{\Omega}_2}{\Gamma, \tilde{\mu} \vdash e_1; e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_1 \cup \tilde{H}_2, \tilde{\Omega}_1 \cup \tilde{\Omega}_2} \quad \text{(TCOMP)}$$

$$\frac{\Gamma, \tilde{\mu} \vdash e_1 : \Gamma', \tilde{\mu}', \tilde{H}_1, \tilde{\Omega}_1 \qquad \Gamma, \tilde{\mu} \vdash e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_2, \tilde{\Omega}_2}{\Gamma, \tilde{\mu} \vdash \text{if } (*) \text{ then } e_1 \text{ else } e_2 : \Gamma' \uplus \Gamma'', \tilde{\mu}[\forall v.(v \mapsto \tilde{\mu}'(v) \cup \tilde{\mu}''(v))], \tilde{H}_1 \cup \tilde{H}_2, \tilde{\Omega}_1 \cup \tilde{\Omega}_2} \quad \text{(TIF-ELSE)}$$

$$\frac{\Gamma[\forall v.(v \mapsto (\Gamma(v).o)^{(\Gamma(v).\tilde{\pi}^j \oplus 1)})], \tilde{\mu} \vdash e : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}}{\Gamma, \tilde{\mu} \vdash \text{while}^j\ (*) \text{ do } e : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}} \quad \text{(TW)}$$

Fig. 7. Typing

(rules 1-4 in Figure 8) needs to consider both allocation sites and abstract loop iteration counts contained in types. If two types $\tilde{\tau}_1$ and $\tilde{\tau}_2$ do not have the same allocation sites o (rule 2), performing join on them yields $\top$. Otherwise, their loop status abstractions $\tilde{\tau}_1.\tilde{\pi}$ and $\tilde{\tau}_2.\tilde{\pi}$ are forced to join (rule 3). Loop labels ($\tilde{\pi}.l$) in the two status pairs have to be the same because they are associated with the same allocation site. Joining ICAs $\tilde{i}_1$ and $\tilde{i}_2$ is shown in rule 4: if $\tilde{i}_1 \neq \tilde{i}_2$, the result is $\top$, meaning that nothing is known about the iteration where the object is created. A finite-height type lattice can be defined based on the operations in Figure 8, with $\top$ and $\bot$ as the maximum and minimum types in the lattice. Types with different allocation sites are not comparable.

In the beginning of each loop iteration (shown in rule TW), the ICA of each type (whose allocation site is under loop j) in the type environment is incremented by using operator $\oplus$, which is defined in Figure 8 (rule 5). The goal of this is to "clear the loop status" of the objects that are carried over from the last iteration, so that these (old) objects and the fresh objects created in the current iteration can be distinguished. Note that a fixed point is computed for the handling of loops: while each iteration of the loop may yield a different solution, the fixed-point solution must not be smaller than this solution.

Next, we explain how to detect data structures whose objects are guaranteed to be allocated in the same loop iteration, using the type and effect system.

Lemma 1. *(*Connected objects created in the same iteration*). For each heap points-to effect $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2 \in \tilde{H}$, if $\tilde{\tau}_1.\tilde{\pi}.\tilde{i} = \tilde{\tau}_2.\tilde{\pi}.\tilde{i} = 1$ for a particular loop j (i.e., $\tilde{\tau}_1.\tilde{\pi}.l = \tilde{\tau}_2.\tilde{\pi}.l = j$), in each iteration of j where an instance of $\tilde{\tau}_1.o$ and an instance of $\tilde{\tau}_2.o$ are connected by a store operation, these instances must be allocated in this same iteration.*

[Join of Γ]

(1) $\Gamma_1 \uplus \Gamma_2 = \Gamma_3$, where $\forall a \in \text{DOM}(\Gamma_3), \Gamma_3(a) = \begin{cases} \Gamma_1(a) & \text{if } a \in \text{DOM}(\Gamma_1) \text{ and } a \notin \text{DOM}(\Gamma_2) \\ \Gamma_2(a) & \text{if } a \in \text{DOM}(\Gamma_2) \text{ and } a \notin \text{DOM}(\Gamma_1) \\ \Gamma_1(a) \uplus \Gamma_2(a) & \text{if } a \in \text{DOM}(\Gamma_1) \cap \text{DOM}(\Gamma_2) \end{cases}$

(2) $\tilde{\tau_1} \uplus \tilde{\tau_2} = \begin{cases} \tilde{\tau_1} & \text{if } \tilde{\tau_2} = \bot \\ \tilde{\tau_2} & \text{if } \tilde{\tau_1} = \bot \\ (\tilde{\tau_1}.o)^{\tilde{\tau_1}.\tilde{\pi} \uplus \tilde{\tau_2}.\tilde{\pi}} & \text{if } \tilde{\tau_1}.o = \tilde{\tau_2}.o \\ \top & \text{otherwise} \end{cases}$

(3) $\tilde{\pi_1} \uplus \tilde{\pi_2} = \langle \tilde{\pi_1}.l \ , \ \tilde{\pi_1}.\tilde{i} \uplus \tilde{\pi_2}.\tilde{i} \rangle$

(4) $\tilde{i_1} \uplus \tilde{i_2} = \begin{cases} \tilde{i_1} & \text{if } \tilde{i_1} = \tilde{i_2} \\ \top & \text{otherwise} \end{cases}$

[Operator $\oplus$]

(5.1) $\tilde{\pi}^j \oplus 1 = \begin{cases} \langle \tilde{\pi}.l \ , \ \tilde{\pi}.\tilde{i} \oplus 1 \rangle & \text{if } \tilde{\pi}.l = j \\ \tilde{\pi} & \text{otherwise} \end{cases}$

(5.2) $\tilde{i} \oplus 1 = \begin{cases} 1 & \text{if } \tilde{i} = 0 \\ \top & \text{otherwise} \end{cases}$

Fig. 8. Auxiliary operations

Proof sketch. Consider a specific iteration k of j. If both objects are allocated in this iteration, their corresponding abstract iteration counts $\tilde{\tau_1}.\tilde{i}$ and $\tilde{\tau_2}.\tilde{i}$ are both updated to 1 upon their creation (rule TNEW). In the very beginning of the next iteration k + 1, $\tilde{\tau_1}.\tilde{\pi}.\tilde{i}$ and $\tilde{\tau_2}.\tilde{\pi}.\tilde{i}$ will be incremented to $\top$ (rule TW) because these objects are carried over from the last iteration. If in this iteration, both of their allocation sites are executed again, the two ICAs (for the two new instances) are set back to 1 (rule TNEW). This process (of setting the ICAs to $\top$ and then 1) is repeated if these allocations are executed during every iteration of j until j terminates. However, if one of the allocation sites (say o_1) is not executed in iteration $k + 1$, its corresponding ICA $\tilde{\tau_1}.\tilde{\pi}.\tilde{i}$ will keep the value $\top$. Hence, at the end of iteration $k + 1$, $\tilde{\tau_1}.\tilde{\pi}.\tilde{i} = \top$ and $\tilde{\tau_2}.\tilde{\pi}.\tilde{i}$ = 1. Because the final solution Γ is a fixed point and $\top$ is greater than any other abstract value, $\top$ will be recorded in Γ for $\tilde{\tau_1}.\tilde{\pi}.\tilde{i}$ even though o_1 may allocate instances again later in the loop.

Note that $\tilde{\tau_1}.\tilde{\pi}.\tilde{i} = \tilde{\tau_2}.\tilde{\pi}.\tilde{i}$ = 1 does not necessarily indicate that $\tilde{\tau_1}.o$ and $\tilde{\tau_2}.o$ are executed in *every iteration* of loop j. Their ICAs are 1 as long as their instances cannot escape from the iteration where they are created to the next iteration of the loop. This feature allows the analysis to report potentially-hoistable data structures even though their construction code (i.e., stores that connect objects in them) is guarded by conditionals.

Similarly, given a dependence effect $\tilde{\tau_1}.f \preceq \tilde{\tau_2}.g$, if $\tilde{\tau_1}.\tilde{\pi}.\tilde{i} = \tilde{\tau_2}.\tilde{\pi}.\tilde{i} = 1$ for a loop j, we can safely conclude that this whole dependence (i.e., computation) chain from $\tilde{\tau_1}.f$ to $\tilde{\tau_2}.g$ occurs in the same iteration of j, because there are only stack variables between the two end heap locations of the chain.

3.2 Hoistable Logical Data Structures

In this subsection, we introduce the notion of hoistable logical data structures based on the points-to and dependence effect abstractions computed by the type and effect system. As discussed earlier, here we address the question "what data is hoistable in the best scenario"—that is, to find hoistable logical data structures that meet the two

criteria discussed in Section 2. Whether and how they can actually be hoisted will be decided by the user upon inspection. This subsection presents mathematical properties of hoistable data structures.

Definition 1. *(*Constrained closures of $\tilde{H}$ and $\tilde{\Omega}$*) Constrained closures of $\tilde{H}$ and $\tilde{\Omega}$ are represented by relations $\trianglerighteq^*_{j,\tilde{i}_1,\tilde{i}_2}$ and $\trianglelefteq^*_{j,\tilde{i}_1,\tilde{i}_2}$, where parameters j, $\tilde{i}_1$, and $\tilde{i}_2$ denote a loop label, a lower bound, and an upper bound of ICAs, used to compute transitive relationships. We define order $\leq$ on the ICA domain $\tilde{i}$ as $0 \leq 1 \leq \top$.*

(1) Closure $\trianglerighteq^*_{j,\tilde{i}_1,\tilde{i}_2}$ (on $\tilde{H}$) selects a set of data structures (whose nodes are types), in which each edge has the form $o_1^{\tilde{\pi}_1} \trianglerighteq o_2^{\tilde{\pi}_2} \in \tilde{H}$, s.t. $\tilde{\pi}_1.l = \tilde{\pi}_2.l = j$, $\tilde{i}_1 \leq \tilde{\pi}_1.\tilde{i} \leq \tilde{i}_2$, $\tilde{i}_1 \leq \tilde{\pi}_2.\tilde{i} \leq \tilde{i}_2$.

(2) Similarly, closure $\trianglelefteq^*_{j,\tilde{i}_1,\tilde{i}_2}$ (on $\tilde{\Omega}$) selects a set of computation chains, in which each edge has the form $o_1^{\tilde{\pi}_1} \trianglelefteq o_2^{\tilde{\pi}_2} \in \tilde{\Omega}$, s.t. $\tilde{\pi}_1.l = \tilde{\pi}_2.l = j$, $\tilde{i}_1 \leq \tilde{\pi}_1.\tilde{i} \leq \tilde{i}_2$, $\tilde{i}_1 \leq \tilde{\pi}_2.\tilde{i} \leq \tilde{i}_2$.

Note that constraint $\tilde{i}_1 \leq \tilde{\pi}.\tilde{i} \leq \tilde{i}_2$ is used to compute these closures: $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2$ (or $\tilde{\tau}_1.f \trianglelefteq \tilde{\tau}_2.g$) is added into the closure $\trianglerighteq^*_{j,\tilde{i}_1,\tilde{i}_2}$ (or $\trianglelefteq^*_{j,\tilde{i}_1,\tilde{i}_2}$) only when the ICAs $\tilde{\tau}_1.\tilde{\pi}.\tilde{i}$ and $\tilde{\tau}_2.\tilde{\pi}.\tilde{i}$ are "between" the specified parameters $\tilde{i}_1$ and $\tilde{i}_2$. For example, the general closures $\trianglerighteq^*$ and $\trianglelefteq^*$ are special cases of their constrained closures when $\tilde{i}_1 = 0, \tilde{i}_2 = \top$, and j is an arbitrary loop label. It is also easy to see that $\trianglerighteq^*_{j,0,0}$ selects data structures whose objects are all created outside loops. Similarly, a data structure selected by $\trianglerighteq^*_{j,0,1}$ is such that its objects can be created both inside and outside loop j, and the set of inside objects in any run-time instance of the data structure are always allocated in the same iteration. Using constrained closures, we give the following definitions.

Definition 2. *(*Disjoint Data Structure *(DDS)) For an allocation site p located in loop j, a data structure (denoted as δ_p^j) rooted at p with respect to loop j is a graph whose edge set is a subset of $\tilde{H}$. Its run-time instances are guaranteed to be disjoint if, for any edge $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2$ of the data structure, there exists a type $\tilde{\tau}$ for p, s.t.*

$$\tilde{\tau}.o = p \ \wedge \ \tilde{\tau}.\tilde{\pi}.\tilde{i} = 1 \ \wedge \ \tilde{\tau} \trianglerighteq^*_{j,1,1} \tilde{\tau}_1 \ \wedge \ \tilde{\tau} \trianglerighteq^*_{j,1,1} \tilde{\tau}_2$$

A DDS contains objects that are reachable from root p and that are created in the loop. Each run-time instance of a DDS is guaranteed *not* to contain any object instance created (1) outside the loop and (2) inside the loop but in different iterations. This is achieved by using constraint $(j, 1, 1)$ for the closure computation.

Lemma 2. *(*Disjointness of DDS instances*) Given two run-time instances of a DDS δ_p^l created by two iterations of a loop, no run-time object exists in both instances.*

Proof sketch. The lemma can be proved by contradiction. Suppose there is a run-time object that exists in both instances of the data structure. At the point it is added into the second data structure instance (created by a later iteration), the abstract loop iteration count for j contained in its type must be $\top$, which is recorded in the abstract points-to effect that represents this addition. This contradicts the fact that δ_p^l is constructed using closure $\trianglerighteq^*_{j,1,1}$, which limits the abstract iteration count for each type to be 1. □

Definition 3. *(*Iteration-Dependent Field*) A field of the form $\tilde{\tau}.f$ is loop-iteration-dependent (LID) with respect to loop j if*

$$\begin{aligned}&(a)\ \exists \tilde{\tau}'.g:\ \tilde{\tau}.f \preceq^*_{j,0,\top} \tilde{\tau}'.g\ \ \wedge\ \ (\tilde{\tau}' = \top\ \ \vee\ \ \tilde{\tau}'.\tilde{\pi}.\tilde{i} = \top)\\ \vee\ &(b)\ \exists \tilde{\tau}'.g:\ \tilde{\tau}'.\tilde{\pi}.\tilde{i} = 0\ \ \wedge\ \ \tilde{\tau}.f \preceq^*_{j,0,1} \tilde{\tau}'.g\ \ \wedge\ \ \tilde{\tau}'.g \preceq^*_{j,0,1} \tilde{\tau}'.g\end{aligned}$$

Determining whether the value of an object field depends on a specific loop iteration requires to inspect abstract dependence effects. As discussed in (condition 2 of) Section 2, a field can be iteration-dependent if (1) it depends on a value read from a field of an unknown object or an object created in an unknown (different) iteration, or (2) it depends on a field of an object created outside the loop (e.g., $\tilde{\tau}'.g$), and this field is involved in a *dependence cycle* (i.e., it can transitively depend on itself).

Lemma 3. *(*Loop-Invariant Data Structure*) A data structure is loop-invariant if for each type τ in the data structure, $\forall \tilde{\tau}.f \in \textsc{Dom}(\tilde{\Omega}) : \tilde{\tau}.f$ is not an LID field.*

Proof sketch. Let us negate the two conditions in Definition 3, that is, a loop-invariant field $o.f$ can depend only on (1) fields of objects guaranteed to be created in the same iteration with o, or (2) fields of objects created outside the loop and not involved in dependence cycles. For (1), the proof can be done by induction on the chain of dependence edges leading to $o.f$. In the base case, fields without any incoming dependence edge must be assigned newly created objects or `null`, and thus must be loop-invariant. For the inductive step, consider the n-th edge along the chain. If the source field of the edge is loop-invariant, the target field of the edge must also be loop-invariant.

For (2), let us first consider a simplified situation where there is only one outside object field $p.q$ (i.e., p is created outside the loop) involved in the dependence chain. Here are two subcases. First, field $o.f$ (o is an object created inside the loop) depends on $p.q$ and $p.q$ is never written in the loop. It is straightforward to see that $p.q$ does not carry any iteration-specific values and thus $o.f$ is loop-independent.

Second, suppose field $p.q$ is written in iteration i with a value v produced in iteration i'. Here i must equal i', because otherwise $o.f$ could depend on a value computed in a different iteration, which contradicts the statement in (1). Since value v cannot depend on $p.q$ (otherwise $p.q$ would depend on itself), it must come only from objects freshly created in this iteration. Based on the proof of case (1), we know that v must be loop-invariant. If $p.q$ is read later in iteration $k > i$ to produce another value v', v' must also be the same across iterations because $p.q$ is invariant. This reasoning can be easily generalized to handle the more complex situation where multiple outside object fields exist in the dependence chain. □

Definition 4. *(*Hoistable Logical Data Structure *(HDS)) If a data structure δ^j_p is a hoistable logical data structure if it is (1) disjoint and (2) loop-invariant.*

A HDS can exhibit exactly the same behavior at run time when it is located inside the loop and outside the loop, under the assumption that the code statements that access this data structure can be safely hoisted. In fact, instead of reporting only completely-hoistable data structures, our analysis identifies, for each logical data structure, its

hoistable part (that is both disjoint and loop-invariant). The analysis eventually ranks all loop data structures based on their hoistability measurements, in order to quantify the likelihood of successful manual hoisting.

3.3 Analysis Algorithm

This subsection briefly discusses our analysis algorithm, which, for the first time, uses a context-free-language (CFL)-reachability formulation to compute the context-sensitive dependence and ICA information. The CFL-reachability formulation enables a demand-driven analysis that can work on each individual object in the loop and explore the points-to and the dependence relationships only for the fields of this object without performing a whole-program analysis. The analysis has three logical components. The first component is a data structure analysis. In order to discover the data structure rooted at an object, this analysis employs a variation of the CFL-reachability formulation of points-to analysis [5], which models both context sensitivity via method entries and exits and heap accesses via object fields read and writes. For each loop in an actual Java program, data structure root objects are first located. To find such root objects, we first consider objects that are created directly in the loop body. Objects that are created in a method (e.g., used as a factory) invoked by a call site in the loop and that can be returned to the loop-containing method are also considered.

Next, for each root object o, our analysis identifies the set of reachable objects and their points-to relationships. In particular, the analysis looks for chains of stores of the form $a_0.f_0 = new\ X; a_1.f_1 = b_0; a_2.f_2 = b_1; \ldots; a_n.f_n = b_{n-1}; b_n = o$, such that (1) the two variables in each pair (a_i, b_i) for $0 \leq i \leq n$ are aliases and (2) the CFL path for this chain contains balanced method entries and exits. If such a chain can be found, the object represented by $new\ X$ is in the data structure rooted at o, because it could potentially be reached from o at runtime through a sequence of field dereferences $f_n.f_{n-1} \ldots f_1.f_0$. Using this formulation, our hoisting analysis can be performed on demand: it can work directly on each loop object, and performs only the work necessary to detect its data structure and to check its hoistability.

The second component of the analysis is a form of data flow analysis that analyzes each loop to perform type inference. An abstract heap (points-to and data dependence) effect is actually the join of data flow facts on all valid paths from the loop entry to the assignment that connects the two entities in the effect. Aliasing relationships are determined by querying the CFL-reachability-based points-to analysis. The third part is a form of data dependence analysis that detects loop-invariant object fields. This analysis traverses backward the def-use chains from each store that writes to a field of an object in a loop data structure, and checks whether the two conditions in Definition 3 hold for the field. A key challenge in computing precise data dependences lies in the handling of data flow via heap locations. Our analysis initially works on top of a context-insensitive points-to analysis: for each load $a = b.f$, we find the set of all assignments $c.f = d$ such that c and b can alias context-insensitively. We next perform refinement on this candidate set using the CFL-reachability formulation of pointer-aliasing to find whether b and c can indeed alias, and if they can, the calling context for $c.f = d$ under which the value flow occurs. This calling context is used to guide the future graph traversal to follow the appropriate entry/exit edges.

4 Computing Hoistability Measurements

In practice, we have observed that only a small number of data structures and statements in a large program can meet both criteria described in Section 3. This is primarily due to their complex usage and the conservative nature of any static analysis algorithm, which must model this complex usage soundly. While fully-automated transformations are desirable and sometimes possible, the usefulness of the static analyses can be increased even further by generalizing them to provide valuable support for programmer-driven manual code transformations.

Previous studies such as [6,7] have demonstrated that, in many cases, manual tuning with developers' insight can be much more effective than fully-automated compiler optimizations. For instance, a programmer may quickly identify that it is problematic to create a 100-field data structure in a loop with only 1 field iteration-dependent, while the sound transformation would give up and terminate silently. To enlist human effort, we must present to them highly-relevant information that can quickly direct them to a problematic area. In this section, we present two metrics that we use to measure *hoistability* of data structures. These measurements are computed based on the two orthogonal relationships (i.e., points-to and dependence) that are described earlier in the paper.

Dependence-Based Hoistability. (DH) The first metric we consider measures the amount of data in a data structure that is constant during the execution of a loop (i.e., the part that complies with rule 2 in Section 2). This dependence hoistability metric is simply defined as an exponential function $DH = s^{n/s}$, which considers two factors: the total number of fields s in a data structure and the number of its loop-invariant (i.e., non-LID, discussed in Def. 3) fields n. The larger s is, the more performance improvement could potentially be achieved by hoisting it. The larger n/s is, the easier it is for a programmer to hoist this data structure. If s is 1, the data structure contains a single field. Even though this field is invariant (i.e., n/s is 1), hoisting it may not have a large impact on performance. If n/s is a small number (i.e., most of its fields are not invariant), the result of $s^{n/s}$ can be very small (i.e., close to 1) regardless of how large s is, which also indicates it is not worth spending time as the data structure may be too difficult to hoist. In addition, we choose an exponential function instead of a polynomial function as the metric because the exponential function "penalizes" cases where n is small, while a polynomial function would be "fair" for all cases of n. For example, if n=s/2 (half the fields are invariant), our exponential function will give the square root of s, while a polynomial function may give a much larger number.

Structure-Based Hoistability. (SH) Similarly to the first metric, the second metric considers, for each data structure, how many objects in it are guaranteed to be allocated in the same iteration (i.e., the part that complies with condition 1 in Section 2). This structural hoistability metric is defined as $SH = t^{m/t}$, where t is the total number of objects in the data structure and m is the number of such objects whose ICA is 1. The value of t is computed by counting the number of objects that are context-sensitively reachable from the root object of a data structure. It is clear to see that SH considers both the size of the data structure and the size of its disjoint part. Note that when m/t is 1, this data structure is a DDS (as discussed in Def. 2), as it is guaranteed to have disjoint instances in all loop iterations.

During our studies, we found that DH is much more useful than SH in distinguishing data structures that are easy to hoist manually from those that are not. First of all, s is a more accurate measurement of the size of a data structure than t, as the data structure can still be large if it contains fewer objects but each object has more fields. Second, we found that for a large number of data structures in our benchmarks, their m/t is 1, which means they are all DDS. It would be quite labor-intensive to inspect all of them and check if they are hoistable. To help the programmer quickly identify truly optimizable data structures, we focus on DDS (whose m/t is 1) and compute dependence-based hoistability (DH) only for these data structures. Finally, only DDS are ranked (based on their DH measurements) and presented to the user for inspection.

Incorporating Dynamic Information. For performance tuning, dynamic (frequency) information is necessary to help programmers focus on "hot" entities (e.g., calling contexts, data structures, etc.). For example, it could be more beneficial to hoist a small, but frequently-allocated data structure than a big, occasionally-occurring data structure. Frequency information can be easily incorporated into the two hoistability metrics. For example, for DH, its definition can be simply extended to $DDH = (f * s)^{n/s}$, where f represents the allocation frequency of the root object of the data structure.

While these metrics are simple, we show that they are effective in locating data structures that are mostly hoistable and easy to optimize. In this work, we focus on demonstrating that the static analyses are useful—even with these simple metrics, the reports can quickly guide us to data structures that are truly optimizable.

5 Evaluation

We implemented the analysis on the Soot analysis framework [8] and evaluated it on the 19 Java programs shown in Table 1. All experiments were conducted on a quad-core machine with an Intel Xeon X3363 2.83GHZ processor, running Linux 2.6.18. The setup for static analysis (similarly to [9]) used the library classes from Sun JDK 1.5.0_06 and 4GB of max heap space. Programs in the table were from the `SPECjvm98`, `Ashes`, and `DaCapo` (its 2006 release and a pre-release) benchmark suites. We did not choose the recent release of DaCapo, because it contains applications making heavy use of class-loading/reflection, which can prevent any static analysis from producing precise information. For `SPECjvm98`, we included only large programs that have loop objects.

5.1 Static Analysis and Hoisting

Table 1 reports statistics of the benchmarks, the analysis, and the dependence-based hoistability measurements. For a GUI application `muffin`, we could not find an appropriate test case to perform profiling, and thus, "-" is used to fill out columns that report dynamic-information-based measurements. We could not perform profiling for `eclipse` either, as different plugins use their own class loaders, making it difficult for them to access our profiling library without modifying their customized class loaders. The cost of the analysis is generally proportional to the number of loop objects processed because of its demand-driven nature. The analysis running time can also be influenced by the size of the code base, as the analysis is context-sensitive and the

Table 1. Shown in the first seven columns of the table are the general statistics of the benchmarks and the analysis: the benchmark names, the numbers of methods (in thousands) in Soot's Spark context-insensitive call graph (*M*), the numbers of loops inspected (*Loops*), the numbers of loop objects considered (*Obj*), the running times of the analysis (*Time*), the total numbers of disjoint data structures (*DDS*), and the total numbers of hoistable logical data structures (*HDS*). Columns *SF* and *SIF* in section DH (i.e., dependence-based hoistability) show the total numbers of fields (*SF*) and the numbers of loop-invariant fields (*SIF*), averaged among the top 10 DDS that we chose to inspect. These data structures are ranked based on the dependence-based hoistability measurement (DH). Columns *DF* and *DIF* report the same measurements as *SF* and *SIF*, except that the inspected data structures are ranked using DDH that incorporates dynamic frequency information.

Benchmark	(a) Analysis statistics						(b) DH			
	#M(K)	*#Loops*	*#Obj*	*Time (s)*	*#DDS*	*#HDS*	*#SF*	*#SIF*	*#DF*	*#DIF*
jack	12.5	88	13	1224	5	3	797	62	797	62
javac	13.4	270	89	1745	33	8	45	31	42	28
soot-c	10.4	475	17	3043	7	3	56	36	56	36
sablecc-j	21.4	202	228	7910	82	53	429	194	221	61
jess	12.8	119	32	304	7	1	1135	51	1135	51
muffin	21.4	318	96	10766	47	8	1503	198	-	-
jflex	20.2	209	17	2325	9	0	55	17	55	17
jlex	8.2	108	9	5549	4	0	36	6	36	6
java-cup	8.4	99	19	474	4	0	107	57	107	57
antlr	12.9	154	3	77	2	1	3	0	3	0
bloat	10.8	562	141	3476	36	10	1536	136	674	46
chart	17.4	482	102	12746	6	0	84	19	84	19
xalan	12.8	17	8	63	6	0	78	24	78	24
hsqldb	12.5	33	10	178	5	0	75	19	75	19
luindex	10.7	14	5	163	5	0	65	15	65	15
ps	13.5	117	21	1784	21	11	36	20	34	20
pmd	15.3	594	30	168	15	2	127	68	127	68
jython	27.5	614	48	423	24	3	77	25	190	26
eclipse	41.0	3322	93	21557	80	52	1182	180	-	-

number of contexts often grows significantly when the size of the program increases. It is clear that the analysis can scale to large applications, including the `eclipse` framework, which has millions lines of code in its implementation.

Across all applications we observe large numbers of disjoint data structures (DDS) and hoistable logical data structures (HDS). This is a strong indication of the existence of optimization opportunities that can be exploited by human experts, which motivates our proposal of computing hoistability measurements to help manual tuning. To compare the effectiveness of *DH* (i.e., dependence-based hoistability measurement proposed in Section 4) and *DDH* (i.e., the profile-based version of it) in finding optimization opportunities, we inspected the top 10 data structures (or the total number of data structures if it is smaller than 10) that appear in both reports. The total numbers of fields / the numbers of loop-invariant fields for these inspected data structures are shown in columns *SF/SIF* and *DF/DIF*, for these two kinds of reports. Profiling is implemented

by Soot-based bytecode instrumentation that records the execution frequency for each loop. The goal of this comparison is to understand how much impact the dynamic information can have on the interpretation of reports. Specifically, can *DDH* (i.e., the incorporation of the run-time frequency f) lower the ranks of data structures that are highly-likely to be optimized (i.e., have larger n/s but smaller f)?

The ratio between *SIF* (or *DIF*) and *SF* (or *DF*) indicates, to a large degree, the difficulty of hoisting data structures manually by inspecting the analysis reports. The larger it is, the easier it may be for a performance expert to modify the data models to hoist them. It is clear that the ratios of *DIF*/*DF* are generally close to those of *SIF*/*SF*. In many cases, the former are even greater than the latter (e.g., `bloat`). This observation indicates that *DDH* can expose not only hot spots (i.e., frequently-allocated objects), but also *optimizable* data structures.

5.2 Case Studies

We have carefully inspected the generated analysis reports (with dynamic information incorporated) for these 19 benchmarks and found optimizable data structures in almost every one of them. This subsection presents five representative case studies, in which either large performance improvement was seen, or interesting bloat patterns were found. These applications are `ps`, `xalan`, `bloat`, `soot-c`, and `sablecc-j`, all with large code base and a great number of loop objects. The performance problems we show in this paper are new and have never been reported by any previous work. Performance improvements are measured on Sun Hotspot 64-bit Server VM build 1.6.0_11.

Through these studies, we found that the analysis is quite useful in helping programmers find mostly-loop-invariant data structures and the execution inefficiencies due to these data structures. It took us about three days to find and fix problems in these five applications, among which we had studied only `bloat` before. Note that most of this time was spent on developing fixes rather than finding data structures that can be easily hoisted: for each benchmark, we looked at only the top 10 data structures in the reports (due to the limited time we had), and found that most of them were indeed hoistable. Even larger optimization opportunities could have been possible if we had inspected more warnings generated by the tool.

It is important to note that it would not be possible to find such problems by using any existing profiling tool: to detect loop-invariant data structures, a purely dynamic analysis has to perform whole-program *value profiling*, a task that is prohibitively expensive for large-scale, long-running Java programs. This is the reason why we have not compared our results with dynamic analysis reports.

ps. `ps` is a postscript interpreter. The top data structure in the list ranked by *DDH* is rooted at a `NameObject` object created in a loop in method `execute` of class `makeDictOperatorDB`. The loop is used to traverse a stack: for each stack element (i.e., a `NameObject` object containing a key and a value), the goal is to remove the '/' character in the key of the element. The way the loop is implemented is that it creates another (backup) stack, pops the original stack, creates a new `NameObject` object with most values copied directly from the original object, and pushes this new object onto the backup stack. Eventually all the new objects in the backup stack are pushed

back onto the original stack. The creation of such `NameObject` objects directs us to think about this implementation, and especially about the way the stack is operated. In fact, this process can be done entirely *in place* so that these objects do not even need to be created. A further inspection of code found an even more interesting problem. The programmer seems to ignore the fact that class `Stack` is a subclass of `List` in JDK and uses `push` and `pop` to implement everything related to stack. For example, this same pop-push pattern is used even for element retrieval. For almost each occurrence of this problematic stack usage pattern, there is a corresponding (mostly loop-invariant) data structure in our report. We removed only two occurrences of such a pattern (in `makeDictOperatorDB.execute` and `DictStack.getValueOf`), and this resulted in a reduction of 82.1% in running time (from 28.3s to 5.3s) and 70.8% reduction in the total number of objects created (from 10170142 to 2969354).

xalan. `xalan` is an XLST processor for XML documents. The problem we found is in a test harness (`XalanHarness`) used by DaCapo to run the benchmark. This harness class creates multiple threads to transform input XML files. In method `run`, there is a `while(true)` loop that assigns jobs to different threads. Our report shows that a data structure rooted at a `Transformer` object created in the loop is a HDS (shown in Figure 1(b)). The same *Transformer* object is created every time the loop iterates, and then used by different threads for transforming the input files. It is highly unlikely to automatically hoist this data structure because this object is created by using a transformer factory object, which is obtained from a reflective call. After hoisting this allocation site, we observed a 10% reduction in running time and 1% reduction in the number of objects created. This problem has also been confirmed by the DaCapo maintainers [10] and will be handled in the next release of the DaCapo benchmark set.

bloat. `bloat` is a program analysis tool designed for Java bytecode-level optimizations. Many loop data structures reported by our analysis are objects of inner classes that are declared exactly at the point where their objects are needed. In `bloat`, most of these objects are created to implement visitor patterns. Hence, the objects are used only for method dispatch and do not contain any data related to the program context under which they are created. These objects commonly exist in loops, and in many cases we found them even located in loops with many layers of nesting. This problem exemplifies a typical object-oriented philosophy: the programmer should focus on patterns and abstractions when coding, and leave the mess to the run-time system. By simply hoisting the reported objects (and the declarations of their classes) out of the loops, we saw 11.1% running time reduction and 18.2% reduction in the number of created objects.

soot-c. `soot-c` is a part of the Soot analysis framework [8] benchmarked in the Ashes benchmarks [11]. One top data structure reported by our analysis is rooted at a `StmtValueBoxPair` object created in a loop (in a constructor of `jimple.SimpleLocalUse`) that builds def-use relationships as follows:

```
while(defIt.hasNext()){
        List useList = (List) stmtToUses.get(defIt.next());
        useList.add(new StmtValueBoxPair(s, useBox));
}
```

For each statement s that uses a variable, the program finds a set of statements that define the variable, creates a `StmtValueBoxPair` object, and adds it to the list. These `StmtValueBoxPair` objects, while containing the same values, are created for safety purposes: if one such pair is changed later, other pairs should not be affected. After inspecting the code, we found that the use list associated with each statement is never changed after the jimple statement chain is constructed for a program. Even if a client analysis could change it by inserting statements, Soot always creates a new object to represent this (newly-established) def-use relationship rather than change the original object. This problem shows a typical example of an over-protective implementation, where several different mechanisms are used simultaneously to enforce the same property while one (or a few) of them may be sufficient to do so. By sharing one `StmtValueBoxPair` object among multiple def statements, we achieved 2.5% running time reduction and 3.5% reduction in the number of created objects. In this example, we can see once again the advantage of tool-assisted manual tuning: this data structure can never be eligible for hoisting from the perspective of any fully-automated analysis. However, the human insight did make hoisting happen as it is unnecessary to have these instances simultaneously.

sablecc-j. `sablecc-j` is a version of the Sable Compiler Compiler that produces the sablecc files (parser, lexer, etc.) for a preliminary version of the jimple grammar. Similarly to the problems found for `bloat`, a large number of HDS reported are related to inner classes: two such classes are declared in `sablecc.GenParser` to perform depth-first traversal of syntax trees, and one such class is declared in `sablecc.DFA` to represent an interval in a char set. Creating multiple objects for each such class is completely unnecessary. Hoisting these class declarations and their objects resulted in 6.7% running time reduction and 2.5% reduction in the number of objects created.

Evaluation Summary. While all loop-invariant data should be hoisted out of loops, the tight data coupling in an object-oriented program makes it impossible for us to do so (either automatically or manually). To help programmers focus on data structures that are (1) easy to hoist and (2) worth hoisting, we propose to compute hoistability measurements. Through these case studies, we demonstrate that our measurements are effective in pinpointing such data structures. In fact, by inspecting reported data structures, we found many performance problems and achieved significant performance improvement. Some invariant data structures that we have managed to hoist are due to (deeper) design issues such as inefficient implementations of design patterns (e.g., visitors in `bloat`) or over-protective implementation strategies (e.g., `soot-c`). Our measurements were also helpful in revealing these issues by exposing their symptoms (i.e., mostly-invariant data structures).

6 Related Work

The related work can be broadly classified into three categories: loop optimizations, runtime bloat detection techniques, and related static analyses.

Loop Optimizations. In the literature on compiler optimization [2], loop optimizations are important techniques that, for example, improve locality and make effective use

of parallel processing capabilities. There is a large body of work on making the execution of loops faster. This set of techniques includes, for example, loop interchange, loop splitting, loop unrolling, loop fusion, loop-invariant code motion, etc. In high-performance computing, loop optimizations play a key role in automated parallelization for exploiting the parallelism capabilities of the hardware. Broader overview and more detailed descriptions of these techniques is available from a number of sources (e.g., [12,13,14]).

Bloat Detection. Mitchell and Sevitsky [15] introduce a way to find data structures that consume excessive amounts of memory. Work by Dufour *et al.* [16] uses a blended escape analysis to characterize the excessive use of temporary objects, which can also be used to help diagnose performance problems. JOLT [17] is a tool that makes aggressive method inlining decisions based on the identification of regions where a large volumes of temporary objects are observed. The approach from [18] dynamically identifies inappropriately-used Java collections to detect bloat. Recent work proposes dynamic analyses [6,7] that detect memory bloat by profiling copy chains and finding low-utility data structures, and static analysis [9] that finds inefficiently-used data structures. A detailed overview of the causes of runtime bloat can be found in [19,20].

Different from these existing techniques, our work focuses on loop-invariant data structures, and aims to help programmers identify them using a series of sophisticated static analyses. As discussed earlier, it may not be feasible to find such optimizable data structures using a dynamic analysis, which requires to profile all values generated during an execution, a task prohibitively expensive for real-world programs. Bhattacharya *et al.* propose an escape-analysis-based static technique [21] that can find reusable collections in a loop. Our analysis is more powerful: we can find general data structures that have disjoint instances as well as the same shapes and data content.

Related Static Analyses. The loop iteration abstraction is first proposed in [3] for computing their conditional must-not-alias properties. Our abstraction extends this approach for the purpose of detecting hoistable data structures, by computing ICA-annotated points-to and dependence relationships. Object inlining [22,23] is a static technique that finds sets of objects that can be efficiently fused into larger objects, and fuses them. While both object inlining and our analysis aim to achieve better performance and need to find objects created in the same control flow region, our analysis targets a different class of performance problems. In addition, our analysis can assist a programmer to do manual tuning, a task that is difficult for object inlining to perform. Gheorghioiu *et al.* propose a static analysis [24] to identify *unitary* allocation sites whose instances are disjoint so that these instances can be preallocated and reused. While this is similar to the detection of disjoint data structures in our work, we can find more opportunities such as data structures with loop-invariant data content and shapes.

Work from [4] presents *recency abstraction*, a technique that distinguishes most-recently-allocated-object (MRAO) and non-MRAO for each allocation site in order to enable strong updates for a points-to analysis. While this is similar to our iteration abstraction that distinguishes objects created in the current iteration and previous iterations, our analysis uses such an abstraction for identifying loop-invariant data structures, instead of improving the precision of a points-to analysis.

There exists a body of reachability analyses that can discover shapes of data structures. Such algorithms range from flow-sensitive approximations of heap shape (e.g., [25,26,27]) to decision procedures (e.g., [28,29]). While our approach can be less precise than these algorithms (especially in handling recursive data structures), its precision may be sufficient to find hoistable data structures. In addition, our demand-driven analysis is more scalable and has been shown to run successfully on large-scale applications including `eclipse`.

Ownership types [30,31,32,33,34,35] provide a way of specifying object encapsulation and enabling local reasoning about program correctness in object-oriented programs. While ownership types may be sufficient to select loop data structures and check whether they are confined, these types cannot detect loop-invariant values, which are dependence-related properties.

7 Conclusions

This paper presents the first static technique that detects loop-invariant data structures. We focus on data models and look for logical data structures that can be hoisted. Instead of transforming the program and hoisting data structures automatically, we propose to measure the hoistability of a data structure: the dependence-based hoistability metric measures the amount of loop-invariant data in a data structure. We have implemented the analyses and presented an evaluation on a set of 19 Java benchmarks. Our experimental results demonstrate that the analysis can scale to large applications and the measurements can be useful in finding significant optimization opportunities.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. This material is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040, grant CCF-1017204, and by an IBM Software Quality Innovation Faculty Award. Guoqing Xu was supported in part by an IBM Ph.D. Fellowship Award.

References

1. Mitchell, N.: Personal communication (2009)
2. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (2006)
3. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL, pp. 327–338 (2007)
4. Balakrishnan, G., Reps, T.: Recency-Abstraction for Heap-Allocated Storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
5. Sridharan, M., Bodik, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI, pp. 387–400 (2006)
6. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: Profiling copies to find runtime bloat. In: PLDI, pp. 419–430 (2009)
7. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., Sevitsky, G.: Finding low-utility data structures. In: PLDI, pp. 174–186 (2010)

8. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
9. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. In: PLDI, pp. 160–173 (2010)
10. DaCapo bug repository: Bloat report, http://sourceforge.net/tracker/?func=detail&aid=2975679&group_id=172498&atid=861957
11. Ashes Suite Collection, http://www.sable.mcgill.ca/software
12. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys 26(4), 345–420 (1994)
13. Wolfe, M.: High performance compilers for parallel computing. Addison-Wesley Publishing Companay (1996)
14. Allen, R., Kennedy, K.: Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann Publishers Inc. (2001)
15. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: OOPSLA, pp. 245–260 (2007)
16. Dufour, B., Ryder, B.G., Sevitsky, G.: A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In: FSE, pp. 59–70 (2008)
17. Shankar, A., Arnold, M., Bodik, R.: JOLT: Lightweight dynamic analysis and removal of object churn. In: OOPSLA, pp. 127–142 (2008)
18. Shacham, O., Vechev, M., Yahav, E.: Chameleon: Adaptive selection of collections. In: PLDI, pp. 408–418 (2009)
19. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to Java runtime bloat. IEEE Software 27(1), 56–63 (2010)
20. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: FoSER, pp. 421–426 (2010)
21. Bhattacharya, S., Nanda, M.G., Gopinath, K., Gupta, M.: Reuse, Recycle to De-bloat Software. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 408–432. Springer, Heidelberg (2011)
22. Dolby, J., Chien, A.: An automatic object inlining optimization and its evaluation. In: PLDI, pp. 345–357 (2000)
23. Lhoták, O., Hendren, L.: Run-time evaluation of opportunities for object inlining in Java. Concurr. Comput.: Pract. Exper. 17(5-6), 515–537 (2005)
24. Gheorghioiu, O., Salcianu, A., Rinard, M.: Interprocedural compatibility analysis for static object preallocation. In: POPL, pp. 273–284 (2003)
25. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS 24(3), 217–298 (1999)
26. Chang, B., Rival, X.: Relational inductive shape analysis. In: POPL, pp. 247–260 (2008)
27. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300 (2009)
28. Lev-Ami, T., Immerman, N., Reps, T., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating Reachability Using First-Order Logic with Applications to Verification of Linked Data Structures. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 99–115. Springer, Heidelberg (2005)
29. McPeak, S., Necula, G.C.: Data Structure Specifications via Local Equality Axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
30. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, pp. 311–330 (2002)

31. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. TOPLAS 29(6), 32 (2007)
32. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA, pp. 292–310 (2002)
33. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL, pp. 213–223 (2003)
34. Heine, D.L., Lam, M.S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: PLDI, pp. 168–181 (2003)
35. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. J. ACM 52(6), 894–960 (2005)

Author Index