Giuseppe Castagna (Ed.)

LNCS 7920 ARCoSS

ECOOP 2013 – Object-Oriented Programming

27th European Conference
Montpellier, France, July 2013
Proceedings

Springer

Lecture Notes in Computer Science 7920

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

Giuseppe Castagna (Ed.)

ECOOP 2013 – Object-Oriented Programming

27th European Conference
Montpellier, France, July 1-5, 2013
Proceedings

Volume Editor

Giuseppe Castagna
CNRS, PPS - Université Paris Diderot
Case 7014, 75205 Paris Cedex 13, France

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39037-1 e-ISBN 978-3-642-39038-8
DOI 10.1007/978-3-642-39038-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013940516

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

This volume contains 29 articles selected among 116 submissions. Each submission was initially allocated to at least three Program Committee members—most submissions were allocated more. In all, 436 review reports were produced—some of which contained multiple reports from distinct reviewers—including reviews contributed by 124 external reviewers. The selection process was organized in six phases as described in Section 3 of the handbook that you can find at the end of this volume. In particular, these phases included a rebuttal phase in which authors had the opportunity to respond to reviews, and a Program Committee meeting that took place in Paris and settled the final program. Of the 116 submissions, 12 were coauthored by members of the Program Committee and underwent a different and stricter selection process described in Section 4 of the handbook: They were assigned at least five Program Committee members; they were discussed on line between the rebuttal phase and Program Committee meeting; they had to satisfy higher acceptance criteria and the decision about them was taken before the Program Committee meeting so as to ensure that acceptance for these article was established in absolute terms and not relative to other submissions. At the end of the process, four of the 12 initial submissions were accepted.

The detailed organization of the whole selection process is described in the handbook that can be found at the end of the volume.

Of the 29 contributions in this volume, two received an award. The Program Committe granted the *Best Paper Award* to "RedCard: Redundant Check Elimination for Dynamic Race Detectors" by Cormac Flanagan and Stephen N. Freund. The Artifact Evaluation Committee granted the *Distinguished Artifact Award* to "QUIC Graphs: Relational Invariant Generation for Containers" by Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan.

The final program included three keynote talks: one by Matthew Parkinson, winner of the 2013 Dahl-Nygaard Junior Award; a second by Oscar Nierstrasz, winner of the 2013 Dahl-Nygaard Senior Award; the third keynote talk was given by Pat Hanrahan, who was invited by the Program Committee. The abstracts of these talks are included at the beginning of this volume.

As a personal note, it was a real honor and privilege for me to chair this committee. The Program Committee members worked hard to produce quality reviews and to provide as much feedback as possible to the authors. Thanks to them the selection process was smooth, pleasant, and carried out in good cheer, and I am sincerely grateful to all of them.

April 2013 Giuseppe Castagna

Organization

ECOOP 2013 was organized by the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), the Centre National de la Recherche Scientifique (CNRS), and the Université Montpellier 2, with the collaboration of the Université Paris Diderot, under the auspices of AITO (Association Internationale pour les Technologies Objets) and in association with ACM SIGPLAN and ACM SIGSOFT.

Conference Chair

Marianne Huchard	Université Montpellier 2, France

Program Chair

Giuseppe Castagna	CNRS, Université Paris Diderot, France

Program Committee

Davide Ancona	Università di Genova, Italy
Paris Avgeriou	University of Groningen, The Netherlands
Alexandre Bergel	University of Chile, Chile
Gavin Bierman	Microsoft Research, UK
Kenneth P. Birman	Cornell University, USA
John Boyland	University of Wisconsin, USA
Perry S. Cheng	IBM Research, USA

Pierre Cointe	École des Mines de Nantes, France
Theo D'Hondt	Vrije Universiteit Brussel, Belgium
Mariangiola Dezani	Università di Torino, Italy
Danny Dig	University of Illinois, USA
Sophia Drossopoulou	Imperial College London, UK
Roland Ducournau	Université Montpellier 2, France
Matthew B. Dwyer	University of Nebraska, USA
Erik Ernst	Aarhus University, Denmark
Martin Gaedke	Chemnitz University of Technology, Germany
Yossi Gil	Technion - Israel Institute of Technology, Israel
Arjun Guha	Cornell University, USA
Philipp Haller	Typesafe, Switzerland
Robert Hirschfeld	Hasso-Plattner-Institut, Germany
Doug Lea	State University of New York, USA
Rustan Leino	Microsoft Research, USA
Ben Livshits	Microsoft Research, USA
Klaus Ostermann	University of Marburg, Germany
Jens Palsberg	University of California at Los Angeles, USA
Frank Piessens	Katholieke Universiteit Leuven, Belgium
Bran Selic	Malina Software Corp., Canada
Frank Tip	University of Waterloo, Canada
Laurence Tratt	King's College London, UK
Jan Vitek	Purdue University, USA
Adam Welc	Oracle Labs, USA

Organization Chairs

Roland Ducournau and Christophe Dony	Université Montpellier 2, France

Workshop Chairs

Olivier Zendra	INRIA Nancy, France
Reda Bendraou	Université Pierre et Marie Curie, France
Damien Cassou	Université Lille 1, France
Stéphane Ducasse	INRIA Lille, France
Thierry Monteil	INSA Toulouse, France

Tutorial Chairs

Naouel Moha	Université du Québec à Montréal, Canada
Jean Privat	Université du Québec à Montréal, Canada
Gergely Varró	Technische Universität Darmstadt, Germany

Artifact Evaluation Chairs

Erik Ernst	Aarhus University, Denmark
Shriram Krishnamurthi	Brown University, USA
Jan Vitek	Purdue University, USA

Doctoral Symposium Chairs

Mireille Blay-Fornarino and Philippe Collet	Université de Nice, France

Research Project Symposium Chair

Isabelle Borne	Universié de Bretagne Sud, France
Ileana Ober	Universié de Toulouse, France

Poster and Demo Chairs

Houari Sahraoui	Université de Montréal, Canada
Bernard Carré	Université de Lille, France
Harald Störrle	DTU Informatics, Denmark

Summer School Chairs

James Noble	Victoria University of Wellington, New Zealand
Jan Vitek	Purdue University, USA

Proceedings Chair

Kim Nguyễn	Université Paris-Sud, France

Web Chairs

Chouki Tibermacine and Clémentine Nebut	Université Montpellier 2, France

Social Events Chairs

Clémentine Nebut and Chouki Tibermacine	Université Montpellier 2, France

Sponsorship and Industrial Relationships Chairs

Jean-Paul Rigault	Université de Nice, France
Abdelhak-Djamel Seriai	Université Montpellier 2, France

Student Volunteers Chairs

Jannik Laval	École des Mines de Douai, France
Floréal Morandat	Université de Bordeaux, France
Petr Spacek	Université Montpellier 2, France

Local Organization Chairs

Elisabeth Grèverie and Justine Landais	Université Montpellier 2, France

Sponsors

Golden

Silver

Bronze

Institutional

External Reviewers

Pieter Agten
Leila Amgoud
Malte Appeltauer
Stephanie Balzer
Véronique Benzaken
Lorenzo Bettini
Robert Bocchino
Robert L. Bocchino
Elisa Gonzalez Boix
Carl Friedrich Bolz
Viviana Bono
Johannes Borgström
Caius Brindescu
Andoni Lombide Carreton
Calin Cascaval
Pierre Caserta
Ilaria Castellani
Nicholas Chen
Laurent Christophe
Olexiy Chudnovskyy
Sylvan Clebsch
Mihai Codoban
Myra Cohen
Charles Consel
Willem De Groef
Benjamin Delaware
Antoine Delignat-Lavaud
Dominique Devriese
Werner Dietl
Isil Dillig
Georges Dupéron
Laura Effinger-Dean
Patrick Eugster
Tim Felgentreff
Jerome Feret
Bruno De Fraine
Paolo G. Giarrusso
Kathryn Gray
Yann-Gaël Guéhéneuc
Rachid Guerraoui
Dries Harnie
Sebastian Heil
Stephen Heumann
Hamidah Ibrahim
Bart Jacobs
Claude Jard
Alan Jeffrey
Malinda Kapuruge
Christian Kästner
Andrew Kennedy
William Knottenbelt
Neelakantan Krishnaswami
Izuru Kume
Giovanni Lagorio
Choonghwan Lee
Benjamin Lerner
Yu Lin
Jens Lincke
Yuheng Long
David H. Lorenz
Qingzhou Luo
Sergio Maffeis
Haohui Mai
Geoffrey Mainland
Jacques Malenfant
Claude Marché
Stefan Marr
Viviana Mascardi
Andrew McVeigh
Wolfgang De Meuter
Tommi Mikkonen
Heather Miller
Floréal Morandat
Jan-Tobias Muehlberg
Clémentine Nebut
Jens Nicolay
Nick Nikiforakis
James Noble
Jacques Noyé
Semih Okur
Tobias Pape
Matthew Parkinson
David J. Pearce
Michael Perscheid
Laure Philips
Pascal Poncelet
Michael Pradel
Polyvios Pratikakis
Jean Privat
Aleksandar Prokopec
Cosmin Radoi
Thierry Renaux
Tillmann Rendel
Gregor Richards
Claudio Russo
Juan Pablo Sandoval Alcocer
Max Schaefer
Lionel Seinturier
Peter Sewell
Jan Smans
Manu Sridharan
Bastian Steinert
Reinout Stevens
Chao Sun
Nicolas Tabareau
Marcel Taeumel
Ewan Tempero
Abideen Tetlay
Francesco Tiezzi
Dan Tofan
Omer Tripp
Aaron Turon
Gias Uddin
Veronica Uquillas
Mohsen Vakilian
Yves Vandriessche
Dries Vanoverberghe
Frederic Vogels
Dimitrios Vytiniotis
Tim Wood
Olivier Zendra
Cheng Zhang
Jianjun Zhao
Elena Zucca

Co-specialization of Hardware and Software

Pat Hanrahan

Stanford University, USA

Abstract. Hardware is becoming increasingly specialized because of the need for power efficiency. One way to gain efficiency is to use throughput-oriented processors (e.g. GPUs) optimized for data-parallel applications; these processors deliver more gigaflops per watt than CPUs optimized for single-threaded programs. On mobile and embedded platforms, where batteries limit the available energy, systems-on-a-chip contain multiple processors along with many specialized hardware units. Since most applications perform many different types of computations, the optimal platform will contain a heterogenous mixture of different types of processing elements.

As the hardware has become more diverse, so have the programming models. This is because the low-level programming models reflect the underlying hardware abstractions. Unfortunately, the diversity in programming models makes it challenging for software developers to use emerging hardware platforms.

The classic method for writing portable programs is to use a general-purpose programming language and an optimizing compiler. Unfortunately, current compilers are not powerful enough for todays platforms. Programs written for one class of machines will not run efficiently on another class of machines.

My thesis is that the only practical method for writing programs for heterogeneous machines is to raise the level of the programming model. In particular, I advocate the use of domain-specific languages (DSLs). I will present the case for using DSLs, show how DSLs can cope with diverse hardware, and outline several areas of programming language research that may lead to better methods for building DSLs. Given the trend towards specialized hardware, it seems natural to co-specialize the software.

I Object, or How I Learned to Stop Worrying and Love OOP

Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Abstract. Object-oriented programming was conceived over 50 years ago, and has consistently proved its value in the construction of complex software systems since the 1980s. Nevertheless, the sentiment that "objects are not enough" is often repeated, and object-oriented programming is commonly bashed by respected computer scientists. We claim that OOP is commonly misunderstood. I describe a personal quest for maintainable software with the help of objects during a period of over 30 years, and argue that we still need to embrace objects if we are to realize the benefits of OOP.

Views on Concurrency Verification

Matthew Parkinson

Microsoft Research

Compositional abstractions underly many reasoning principles for concurrent programs: the concurrent environment is abstracted in order to reason about a thread in isolation; and these abstractions are composed to reason about a program consisting of many threads. For instance, separation logic uses formulae that describe part of the state, abstracting the rest; when two threads use disjoint state, their specifications can be composed with the separating conjunction. Type systems abstract the state to the types of variables; threads may be composed when they agree on the types of shared variables.

In this talk, I will present the "Concurrent Views Framework" [1], a metatheory of concurrent reasoning principles. The theory is parameterised by an abstraction of state with a notion of composition, which we call *views*. The metatheory is remarkably simple, but highly applicable: the rely-guarantee method, concurrent separation logic, concurrent abstract predicates, type systems for recursive references and for unique pointers, and even an adaptation of the Owicki-Gries method can all be seen as instances of the Concurrent Views Framework. Moreover, our metatheory proves each of these systems is sound without requiring induction on the operational semantics.

Reference

1. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: Compositional Reasoning for Concurrent Programs. In: Proceedings of POPL (2013)

Table of Contents

Aspects, Components, and Modularity

Types

Language Design

Concurrency, Parallelism, and Distribution

Analysis and Verification 1

Analysis and Verification 2

Modelling and Refactoring

Testing, Profiling, and Empirical Studies

Implementation

CoCo: Sound and Adaptive Replacement of Java Collections

Guoqing Xu

University of California, Irvine, CA, USA

Abstract. Inefficient use of Java containers is an important source of run-time inefficiencies in large applications. This paper presents an *application-level* dynamic optimization technique called CoCo, that exploits *algorithmic advantages* of Java collections to improve performance. CoCo dynamically identifies optimal Java collection objects and safely performs run-time collection replacement, both using pure Java code. At the heart of this technique is a framework that *abstracts* container elements to achieve efficiency and that *concretizes* abstractions to achieve soundness. We have implemented part of the Java collection framework as instances of this framework, and developed a static CoCo compiler to generate Java code that performs optimizations. This work is the first step towards achieving the ultimate goal of automatically optimizing away semantic inefficiencies.

1 Introduction

Large-scale Java applications commonly suffer from severe performance problems. A great deal of evidence [1,2,3,4,5] suggests that these problems are primarily caused by *run-time bloat* [6]—excessive memory usage and computation to accomplish simple tasks—often from inappropriate design/implementation choices (e.g., creating general APIs to facilitate reuse, using general implementations without any specialization, etc.) that exist throughout the program.

Container Inefficiencies. Previous studies [4,7,8,3,9,10] have found that an important source of run-time bloat is the inefficient use of container implementations. Standard libraries of object-oriented languages such as Java and C# contain collection frameworks that provide with users, for each abstract data type (such as `List`), many different implementations (such as `ArrayList` and `LinkedList`), each of which features a different design choice suitable for a specific execution scenario. However, in real-world development, choosing the most appropriate container implementation is challenging. As a result, developers tend to keep using the implementations that are most general or well-known (e.g., `HashSet` for `Set`), regardless of whether or not they fit the usage context. This is especially true in object-oriented programming, as developers can easily write code without deeply understanding implementation details of libraries, and the culture of object-orientation itself encourages generality and quick reuse of library functions. Inappropriate choices of container implementations can lead to significant performance degradation and scalability problems. More seriously, such bottlenecks can never be detected and removed by an existing optimizing compiler that is unaware of these different design principles and trade-offs.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 1–26, 2013.

Run-Time Container Replacement to Achieve Efficiency. We propose a novel container optimization technique, called CoCo, that is able to (1) determine at run time, for each container instance (e.g., a `LinkedList` object) used in the program, whether or not there exists another container implementation (e.g., `ArrayList`) that is more suitable for the execution; and (2) automatically and safely switch to this new container implementation (e.g., replace the old `LinkedList` object with a new `ArrayList` object *online*) for increased efficiency. While there exists work (such as Chameleon [7] and Brainy [9]) that could identify Java collection inefficiencies and report them to users for *offline* inspection, none of these techniques can change implementations *online*. An online approach outperforms an offline approach in the following two important aspects. First, a real-world execution is often made up of multiple phases, each with a different environment processing different types of data. Much evidence shows [11,12] that it is difficult to find a solution that is optimal for the entire execution. Our experimental results also demonstrate that many containers in large applications experience multiple switches during execution. This calls for novel techniques that can change implementations immediately after they become suboptimal. Second, an offline approach relies completely on the developer's effort to fix the detected problems while an online approach shifts this burden to the compiler and the run-time system.

Challenge 1: How to Guarantee Safety. Developing an online technique is significantly more difficult and constitutes the main contribution of this paper. Switching container implementations can easily cause inconsistency issues, leading to problematic executions or even program crashes. To illustrate, consider again the previous example where we replace a `LinkedList` object with an `ArrayList` object. Any subsequent invocation of a `LinkedList`-specific method (e.g., `getLast`) or type comparison (e.g., *o* `instanceof LinkedList`) can either cause the program to fail or change the semantics arbitrarily.

CoCo uses a combination of manually-modified library code and automatically-generated glue code (both are pure Java code) to perform optimizations. Unlike a traditional systems-level (blackbox) dynamic optimization technique that is completely separated from the applications being optimized, CoCo (1) advocates a methodology that can be used by library designers to create optimizable container classes, and (2) provides a static compiler to generate glue code that performs run-time optimizations on these optimizable container classes. All optimizations are performed at the application level within the (manually modified and automatically generated) Java classes, which encode human insight to direct optimizations.

CoCo stands for *co*ntainer *co*mbination—for each container object (whose type is supported by CoCo) created in a Java program, we additionally create a group of other container objects to which this particular object may be switched at run time. This group, together with the original container, is referred to as a *container combo* henceforth. CoCo initially uses the original container object as the active container, leaving this group of associated containers in the inactive state. All operations are performed only on the active container object. Upon adding a new element, the *concrete* element object is added only into the active container while CoCo creates an *abstraction* for this element and adds the abstraction to all inactive containers. This abstraction is actually a placeholder from which we can find the concrete object(s) it corresponds to.

Once a container switch occurs, an inactive container (e.g., a) that contains element abstractions is activated (and the originally active container b becomes inactive). If later a retrieval operation on a locates an abstraction, this abstraction is *concretized* by finding its corresponding concrete element (from b) to guarantee the safety of the subsequent execution. Section 4 discusses how to instantiate this framework to implement `List`, `Map`, and `Set` combos.

Challenge 2: How to Reduce Run-Time Overhead. It is obvious to see that naively creating container groups and performing abstraction-concretization operations can introduce a great amount of overhead. As our ultimate goal is to improve performance, it is equally important for CoCo to reduce overhead as well as ensuring safety. We employ three important optimization techniques to tune CoCo's performance: (1) drop inactive containers once we observe no optimization is possible; (2) use sampling to reduce profiling overhead; and (3) perform lazy creation of inactive containers.

Note that the performance benefit that CoCo may bring to a program results primarily from the *algorithmic advantage* of the selected implementation over the source implementation in an execution environment that favors the former, instead of from other factors such as a more compact design of the data structure or improved cache locality. In fact, performing online container replacement can, in many cases, increase memory space needed (as more objects are created) and/or hurt locality (as concrete elements can be separated in different containers). In order to improve the overall performance, we have to carefully select the container types to be optimized and evaluate various container replacement decisions to find the appropriate configurations under which the performance benefit outweighs the negative impact of the online replacement.

Section 2 gives an overview of how CoCo performs optimizations on containers and Section 3 presents a formalism that shows the general methodology to perform sound container replacement. Section 4 describes our implementations of the CoCo `List`, `Map`, and `Set`. Section 5 discusses our optimization techniques. We have evaluated CoCo on both a set of micro-benchmarks and a set of 5 real-world large-scale Java applications from which container bloat has been found previously. Our experimental results are reported in Section 7. Although we modify only a small portion of the Java Collections Framework, our experimental results show that CoCo is useful in optimizing micro-benchmarks (e.g., the speedup can be as large as 74×). For the 5 large-scale, real-world programs, CoCo is still effective—an average 14.6% running time reduction has been observed. Large opportunities are possible if more (both Java and user-defined) container classes can be modified to support CoCo optimizations.

The contributions of this work are:

- A general methodology to perform sound online container replacement for increased execution efficiency;
- an instantiation of this methodology in the Java Collections Framework that offers automated optimizations for 7 `Set`, `Map`, and `List` containers;
- three optimization techniques developed to reduce replacement overhead;
- an experimental evaluation of CoCo on both a set of micro-benchmarks and a set of real-world benchmarks; the results suggest that CoCo is effective and may be used in practice to optimize container usage.

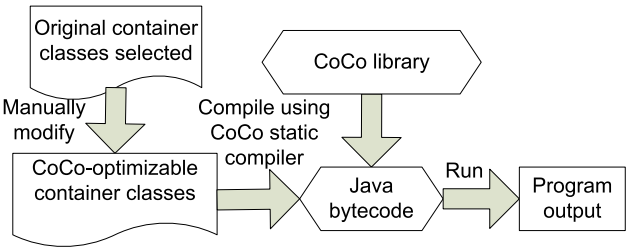


Fig. 1. An overview of the CoCo system

2 Overview

This section uses a `List` example to illustrate how CoCo performs container optimizations. The current version of CoCo focuses on Java built-in container classes, while the CoCo methodology can be applied to optimize arbitrary user-defined containers.

2.1 General Methodology

CoCo performs same-interface optimizations—given a container class that implements interface `I`, CoCo looks for potentially switchable candidates only within the types that implement `I`. Performance gains may be seen sometimes from switching implementations across different interfaces. For example, work from [7] shows that it can be beneficial to switch from $\mathrm{ArrayList}$ to $\mathrm{LinkedHashSet}$ in certain cases. It would be easy to extend CoCo to support multi-interface switches— the developer may need to create a wrapper class that serves as an adapter between interfaces. This class implements APIs of the original interface using methods of the new interface. We leave the detailed investigation of this approach to the future work.

From a set of all container classes that implement the same interface, we select those among which the online replacement may result in large performance benefit (at least large enough to offset the replacement overhead). In this paper, we focus on containers that have clear algorithmic advantages (e.g., lower worst-case complexity) over others in certain execution scenarios. For example, switching from a `LinkedList` to an `ArrayList` upon experiencing many calls to method `get`(i) may reduce the complexity of `get` from O(n) (where n is the size of this `List`) to O(1). This may have much larger benefit than switching from `ArrayList` to `SingletonList` (upon observing there is always one single element)—in this case, no significant algorithmic advantage can be exploited and the benefit resulting from space reduction may not be sufficient to offset the overhead of creating and maintaining multiple containers.

Figure 1 gives an overview of the CoCo system. For the selected container classes, we first modify them manually to add abstraction-concretization operations. The CoCo static compiler then generates glue code that connects these modified classes, performs run-time profiling, and makes replacement decisions. Next, both the generated glue classes and the modified container classes are compiled into Java bytecode, which is executed to enable optimizations. The entire container replacement task is achieved in

```
1 package coco.util;
2 class LinkedList extends java.util.LinkedList
3                     implements OptimizableList{
4   ListCombo combo;  // the combo object
5   LinkedList() { ... }  // the original constructor
    // constructor for CoCo
6   LinkedList(ListCombo c) {
7       this();
8       if(c == null) combo = new ListCombo(this);
9       else combo = c;
10  }
11
12  void add(Object o) {
13      if(combo == null)  this.add$CoCo(o);
14      else               combo.add(o);
15  }
16  // this used to be add(Object o)
17  void add$CoCo(Object o) {/* add obj o*/}
18
19  Object get(int i) {
20      if(combo == null)  return this.get$CoCo(i);
21      else               return combo.get(i);
22  }
23  //this used to be get(int i)
24  Object get$CoCo(int i) { ... }
25}

26 Interface OptimizableList{
27   void add$CoCo(Object o);  Object get$CoCo(); …
28   void addAbstractElement(Object o, int containerID);
29   void replaceConcreteWithAbstract(Predicate p,
                                      int containerID);
30 }
```

(a) CoCo-optimizable **LinkedList** (manually modified from **java.util.LinkedList**) and interface **OptimizableList**

```
1 package coco.util;
2 class ListCombo implements List {
3     OptimizableList active;  OptimizableList[] inactive;
4     ListCombo(OptimizableList l){
5         active = l;
6         createInactiveLists();
7     }
8     OptimizableList[]  createInactiveLists(){
9         inactive = new OptimizableList[...];
10        ...
11        inactive[i] = new ArrayList(this);
12    }
13    void add(Object o) {
14        doProfiling(OPER_ADD);
15        active.add$CoCo(o);
16    }
17    Object get(int i) {
18        doProfling(OPER_GET);
19        return active.get$CoCo(i);
20    }
21    void doProfiling(int operation){
22        incCounter(operation);
23        if(active instanceof LinkedList &&
24                    NUM_GET > THRESHOLD) {
25        //Let's switch to ArrayList
26        // i is the index of the ArrayList object in inactive
27          swap (active, inactive[i]);
28        }
29        ...
30    }
31}
```

(b) Glue class **ListCombo** that does profiling and makes replacement decisions (automatically generated by the CoCo static compiler)

Fig. 2. Examples of a CoCo-optimizable `LinkedList` obtained by manually modifying `java.util.LinkedList` and a combo class generated automatically by the CoCo static compiler

these (manually modified and compiler generated) Java classes at the application level, not in the run-time system.

2.2 Creating CoCo-optimizable Container Classes

Lines 1–25 in Figure 2 (a) show a skeleton of the CoCo-optimizable `LinkedList`, obtained from modifying the original `LinkedList` class in the Java Collections Framework. For ease of presentation, all examples shown in this section are simplified from the real container classes that CoCo uses. In addition, these examples are created only to illustrate how CoCo performs online container replacement, without considering how much overhead the code could incur. We will discuss various overhead reduction techniques in Section 5.

In order to be optimized by CoCo, the modified `LinkedList` must implement the CoCo-provided interface `OptimizableList`, whose skeleton is shown in lines 26–30. This interface declares a set of `X$CoCo` methods, each of which corresponds to a method `X` declared in interface `List` (where `X` can be `get`, `add`, `remove`, etc.). It additionally declares two methods (`addAbstractElement` and `replaceConcrete-WithAbstract`) that will be implemented to perform

abstraction-concretization operations. An X$CoCo method has exactly the same signature as method X. For each X in the original version of LinkedList, we move its entire body into X$CoCo, which, thus, becomes the method that implements the functionality of X. Method X will be used to choose container implementations at run time.

Container Combo. Each CoCo-optimizable container class has an associated combo object of type ListCombo (line 4 in Figure 2 (a)), responsible for profiling container operations and making replacement decisions. Figure 2 (b) shows a skeleton of the glue class ListCombo, which is generated automatically by the static CoCo compiler, given a container interface (e.g., List) and a selected set of its implementing classes to be optimized (e.g., ArrayList and LinkedList).

ListCombo also implements List, and thus, it contains all methods (X) declared in List. Each X in ListCombo is implemented as a *method dispatcher*. To illustrate, consider method add (line 13 in Figure 2 (b)). It first records that this is an ADD operation (line 14), and then invokes method add$CoCo on the currently active list active (line 15). Note that method add$CoCo in different container implementations can be invoked by simply updating the field active during execution.

Example. Consider again class LinkedList in Figure 2 (a). For each constructor in the original version of LinkedList, we add a new constructor that has one additional parameter of type ListCombo (line 6 in Figure 2 (a)). In a client program that calls the original constructor (at line 5), our JVM will replace this call site with a call to the new constructor and pass null as the argument. Upon receiving null, this constructor creates a new ListCombo (line 8) that, in turn, creates a group of other container objects (line 8-12 in Figure 2 (b)) to which this LinkedList object may be switched at run time. The LinkedList object becomes the first active List in this combo (line 5 in Figure 2 (b)). When an inactive container is created, the combo object is also passed into its constructor (line 11 in Figure 2 (b)), and thus, this combo object will be shared among all the candidate containers.

The combo is essentially a central method dispatcher—whenever method add (e.g., in LinkedList) is invoked from a client, it calls method add on the combo object (line 14 in Figure 2 (a)), which, in turn, invokes method add$CoCo on the currently active List object (line 15 in Figure 2 (b)). If, at a certain point during execution, the profiling method (line 21-30 in Figure 2 (b)) observes that the active List is a LinkedList and the total number of get(i) operations invoked is greater than a threshold value (line 23), it changes the active container to ArrayList—the only thing that needs to be done is to find the ArrayList object in the inactive container array and make it active (line 27 in Figure 2 (b)). An illustration of the combo structure can be found in Figure 3.

Preserving Semantics. From a client's perspective, using this combination of containers has no influence on the behaviors of the original LinkedList. The client code always interfaces with the LinkedList methods, although the actual "service" could be provided by a different container object. This design guarantees type safety of the forward execution—we make the modified LinkedList a subclass of the original Java LinkedList (line 2 in Figure 2 (a)), and thus, no failure would result from

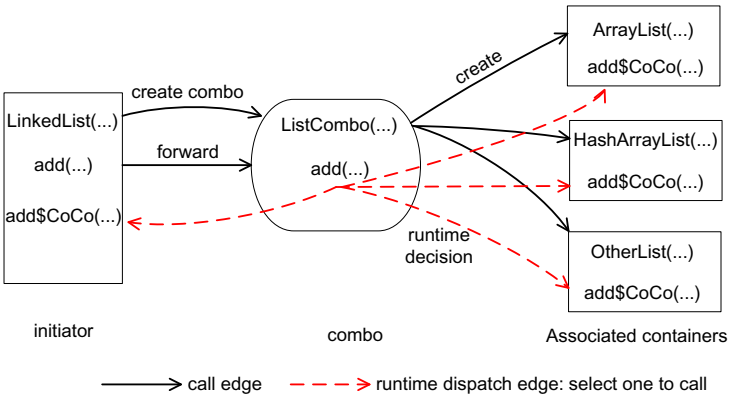


Fig. 3. Graphical illustration of a container combo

future operations that rely on the original type `LinkedList`, such as `instanceof` and the invocation of `LinkedList`-specific methods (e.g., `getLast`).

While manual work is needed to turn a Java (or user-defined) container into a CoCo-optimizable container, much of the original code can be reused and the user needs to add only a few methods, such as `X$CoCo`. The entire class `ListCombo` (except `doProfiling`) is generated automatically by the CoCo compiler. Furthermore, the intended users of CoCo are library designers and implementers, and thus, no extra effort is needed from the end developers. For method `doProfiling` that makes run-time replacement decisions, the condition code (e.g., lines 23-28) can be either generated from a set of user-specified rules (such as those used by Chameleon [7]), or filled in by human experts manually. The current version of CoCo uses the latter approach, while it is easy to develop a rule engine to translate user-defined rules into code predicates. If the designer wishes to combine implementations with conflicting specifications (e.g., some accept `null` values while others do not), she may need to explicitly insert checks in the combo code to direct the dispatch. For example, a `null` value cannot be inserted into a container that does not handle it.

2.3 Using Abstraction and Concretization to Ensure Safety

When an inactive container is activated, all container operations will be performed on it. Key to providing soundness guarantee is, thus, to make sure that any container, if becoming active, can provide service correctly. A natural idea is to move all elements from one container to another upon a switch. However, significant run-time overhead may result from regularly moving elements around containers. If the switch decision is inappropriate and we need to switch back to the original container, moving all elements back and forth can create a great amount of redundant work. To achieve efficiency, we propose an *on-demand* approach—an element is moved from an inactive container to an active container only when it is requested by the client. If the switch decision

```
1 class LinkedList implements OptimizableList {
2     Entry first, last;
3     ...
4     void add$CoCo(Object o){
5         insert(o);
6         for(OptimizableList l : combo.inactiveLists()){
7             l.addAbstract(o, ID_LINKED_LIST);
8         }
9     }
10    Object get$CoCo(int i) {
11        Object o = findEntry(i);
12        if(o instanceof AbstractElement) {
13            AbstractElement ae = (AbstractElement)o;
14            Object[] concreteObjs =
                  ae.concretize(ID_LINKED_LIST);
15            replaceAndExpand(i, concreteObjs);
16            o = findEntry(i);
17        }
18        return o;
19    }
20    void addAbstractElement
                  (Object o, int containerID){
21        AbstractElement ae = createAbstraction
                                  (o, containerID);
22        insert(ae);
23    }
24    ...
25 }
26 class AbstractElement{
27    ListCombo combo;  int targetContainerID;
28    Predicate p;
29    Object[] concretize(int newContainerID) {
30        OptimizableList l = combo.getRuntimeList
                                  (targetContainerID);
31        Object[] objs = new Object[p.size()];
32        for(Integer index : p.iterateAll())
33        { objs[...] = l.get$CoCo(index); }
34        l.replaceConcreteWithAbstract
                  (p, newContainerID);
35        return objs;
36    } ...
37 }
```

Fig. 4. An abstraction-concretization example in LinkedList

is appropriate, the active container will remain active for a relatively long time and concrete elements will be gradually moved to this container as more operations are performed.

To do this, when an ADD operation is performed, it is insufficient to add the incoming object only into the active container—we have to additionally record some information of this element in all inactive containers, so that if any one of them becomes active later and this element is requested by a client, the container would have a way to find it and return it to the client. This information is referred to as the *abstraction* of this concrete element (or sometimes *abstract element*). If an abstraction is found during the retrieval of an element, this abstraction is *concretized* to provide the concrete element(s) it represents. For different types of containers (e.g., `Map`, `Set`, `List`, etc.), we may need to create different types of abstractions. Section 3 discusses a general methodology to create abstractions.

Example. Figure 4 shows part of the modified `LinkedList` responsible for element abstraction and concretization. Highlighted methods are declared in interface `OptimizableList`. After inserting a concrete element *o* into the list (line 5), method `add$CoCo` adds an abstraction of *o* into all inactive lists (lines 6–8). Once an abstraction is found in a retrieval (lines 12–17), it is concretized to get an array of concrete elements it represents (line 14), which are then inserted into the list to replace this abstraction.

Lines 26–36 of Figure 4 show class `AbstractElement`, which is the abstraction that CoCo uses for `List`. Here an abstraction has three pieces of information (lines 27–28): the ID of the container object where the concrete element(s) represented by this abstraction are located (i.e., `targetContainerID`), the combo object from which this container object can be obtained (i.e., `combo`), and a `predicate` (i.e., `p`) that specifies the exact locations of the concrete elements in their host container.

Predicate. A predicate is defined based on how elements are added into and retrieved from a container. For example, for `List`, elements are added and retrieved

based on their *indices*, and thus, the predicate in each element abstraction is defined over the domain of indices (integers), that is, each predicate represents a range of indices (e.g., [0, 15]). When an abstraction is concretized (line 14 in Figure 4), all the elements whose indices satisfy the predicate of the abstraction (lines 32–33) are retrieved from their host container in order to fulfill the client's request. These concrete elements are then moved into this active `List` to avoid future concretization operations (line 15). In addition, we need to remove these concrete elements from their old host container by creating an abstraction there to represent them (line 34). If CoCo switches back to this (old) container later, the same (concretization) process will be performed to retrieve the elements.

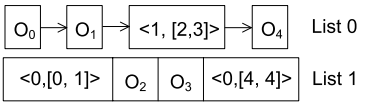


Fig. 5. A `List` combo in which both containers have abstractions

Figure 5 shows an example `List` combo resulting from the code shown in Figure 4. In this combo, both the `LinkedList` (List 0) and the `ArrayList` (List 1) have abstractions (i.e., placeholders) and concrete elements. For example, the abstraction $\langle 1, [2, 3]\rangle$ is the placeholder of the No.2 and No.3 elements and it specifies that these two elements are currently in List 1. Different abstractions can be defined by creating different `AbstractElement` classes (in Figure 4). Section 4 discusses the abstractions that CoCo currently uses for `List`, `Map`, and `Set`, while the general methodology proposed in this paper can be applied to create abstractions for all (Java built-in or user-defined) data structures in order to enable this optimization.

Calls to methods that are specific to each container implementation (i.e., that are not declared in the container Interface) cannot be forwarded to the combo object. For example, if method `getLast` is invoked on a `LinkedList` object after the active container in the combo becomes an `ArrayList`, this call cannot be performed on the `ArrayList` because `getLast` does not exist in `ArrayList`. In this case, `getLast` is still invoked on the `LinkedList` object, but we need to modify this method and concretize an abstraction to bring the last concrete element back to the `LinkedList` to fulfill the request (e.g., if the last element is not the `LinkedList`). It is subject to the developer how the concretization should be performed.

3 Formal Descriptions

This section formalizes the notion of a container, and in that context, formally describes CoCo's core idea on run-time container replacement. Although CoCo currently supports only a few Java built-in containers, this formal framework defines a methodology that can be used to switch arbitrary container implementations. Any instantiation of this framework is guaranteed to be sound.

Definition 1. *(*Container*) A container* Σ *is a triple* $\Sigma = \langle \mathsf{X}, \prec, f\rangle$*, where* X *is a set of concrete elements and* $\prec$ *is a partial order on* X*.* $\prec$ *is determined by a property encoding function* $f(x)$ *that maps each* $x \in \mathsf{X}$ *to an integer* i_x *that encodes a certain property of* x*. For any two elements* $x_1, x_2 \in \mathsf{X}$*,* $x_1 \prec x_2$ *iff.* $f(x_1) < f(x_2)$*.*

Partial order $\prec$ determines how elements are stored in a container. x_1 precedes x_2 in the underlying data structure of the container if $x_1 \prec x_2$. The position of each element depends on a certain property of this element used by the container, which is encoded by function f. For example, for `List`, f maps each x to a unique index; for `HashMap`, f is a hash function that maps each x to the index of the hash bucket where x should be stored based on x's hashcode; for `TreeMap`, f maps each x to the index of x in a list of all elements in the map sorted by the results of their `compareTo` method. This index determines when x will be reached in an in-order traversal of the binary search tree used by `TreeMap`. If neither $x_1 \prec x_2$ nor $x_2 \prec x_1$, there is no constraint regarding the positions of x_1 and x_2. For example, if the hash function (in a `HashMap`) maps both x_1 and x_2 to the same bucket slot, the way they are stored is subject to the specific implementation of `HashMap`.

Definition 2. *(*CoCo-optimizable container*) A CoCo-optimizable container* Γ *is a six-tuple* $\Gamma = \langle \mathsf{X}, \mathsf{A}, \prec, f, \alpha, \gamma \rangle$*, where* X *is a set of concrete elements,* A *is a set of abstract elements,* $\prec$ *is a partial order on* $\mathsf{X} \cup \mathsf{A}$*,* f *is a property encoding function,* α*:* $\mathsf{X} \to \mathsf{A}$ *is an abstraction function that maps each concrete element to an abstraction, and* γ*:* $\mathsf{A} \to 2^{\mathsf{X}}$ *is a concretization function that maps each abstract element to a subset of concrete elements it represents. Partial order* $\prec$ *is redefined as follows:*

(a) $\forall x_1, x_2 \in \mathsf{X}$*:* $x_1 \prec x_2 \Leftrightarrow f(x_1) < f(x_2)$
(b) $\forall a_1, a_2 \in \mathsf{A}$*:* $a_1 \prec a_2 \Leftrightarrow \mathsf{max}_{x \in \gamma(a_1)}(f(x)) < \mathsf{min}_{y \in \gamma(a_2)}(f(y))$
(c) $\forall a \in \mathsf{A}, x \in \mathsf{X}$*:* $a \prec x \Leftrightarrow \mathsf{max}_{y \in \gamma(a)}(f(y)) < f(x)$
(d) $\forall a \in \mathsf{A}, x \in \mathsf{X}$*:* $x \prec a \Leftrightarrow f(x) < \mathsf{min}_{y \in \gamma(a)}(f(y))$

Each CoCo-optimizable container (such as the `LinkedList` shown in Figure 2 (a)) is a mixture of concrete and abstract elements. The (new) partial order $\prec$ is defined on the union of these two groups of elements. The definition of $\prec$ does not change when two concrete elements are compared (i.e., shown in condition (a)). An abstraction $a_1 \prec$ another abstraction a_2 only when the "greatest" concrete element abstracted by $a_1 \prec$ the "smallest" concrete element abstracted by a_2 (i.e., shown in (b)). (c) and (d) show comparisons between an abstraction a and a concrete element x: $a \prec x$ only if the "greatest" concrete element a abstracts $\prec x$. The abstraction function α and the concretization function γ are user-defined and are specific to each type of container optimized by CoCo.

Here the concrete element set X and the abstract element set A are separated for ease of presentation and formal development. In our implementation, they are stored together in the underlying data structure (e.g., the data array for `ArrayList`, the bucket array for `HashMap`, etc.) of a container that used to store only concrete elements. An abstract element in a container is stored in the location where its corresponding concrete element(s) would have been stored if they were in this container.

Definition 3. *(*Contiguous abstraction*) An abstraction* $a \in \mathsf{A}$ *in a container object is a contiguous abstraction iff.* $\forall x_1, x_2 \in \gamma(a)$*: (*$\nexists x \in \mathsf{X}$*:* $x_1 \prec x \prec x_2) \wedge (\forall a' \in \mathsf{A} : a' \neq a \Rightarrow \nexists y \in \gamma(a') : x_1 \prec y \prec x_2)$*.*

A contiguous abstraction represents a range of concrete elements that should be stored contiguously in the container. No other concrete element either directly in the container or abstracted by another abstraction can be put in the middle of them. Allowing only contiguous abstractions in a container simplifies significantly the implementations of the abstraction function α and the concretization function γ. This definition also implies that different contiguous abstractions do not overlap.

Definition 4. *(*Well-formed container*)* *A CoCo-optimizable container* $\Gamma = \langle \mathsf{X}, \mathsf{A}, \prec, f, \alpha, \gamma \rangle$ *is a well-formed container, if (1)* $\forall a \in \mathsf{A}$*: a is a contiguous abstraction and (2)* $\forall x \in \mathsf{X}$*:* $\nexists a \in \mathsf{A}$*:* $x \in \gamma(a)$.

Well-formedness has two important aspects: all abstractions are contiguous abstractions and no abstraction abstracts a concrete element that exists in the container (i.e., an abstraction abstracts only concrete elements located in other containers). It is easy to see that a container with no abstraction is a well-formed container; so is a container with one single abstraction that abstracts all elements.

Definition 5. *(*Well-formed container combo*)* *A container combo* C *is an n-tuple* $\mathsf{C} = \langle \Gamma_1, \Gamma_2, \ldots, \Gamma_{n-1}, i \rangle$*, where* $\Gamma_1, \ldots, \Gamma_{n-1}$ *are CoCo-optimizable containers and* $i \in [1, n\text{ -}1]$ *is the index of the active container.*

A container combo is well-formed iff.

- $\forall j \in [1, n\text{ - }1]$*,* Γ_j *is well-formed; and*
- $\forall j, k \in [1, n\text{ - }1]$*,* $j \neq k$*: (*$|\mathsf{X}_j| + \sum_{a \in \mathsf{A}_j} |\gamma_j(a)| = |\mathsf{X}_k| + \sum_{a \in \mathsf{A}_k} |\gamma_k(a)|$*); and*
- $\forall j \in [1, n\text{ - }1]$*:* $\forall x \in \mathsf{X}_j$*:* $\forall k \in [1, n\text{ - }1]$*:* $k \neq j \Rightarrow \exists a \in \mathsf{A}_k$*:* $x \in \gamma(a)$

A well-formed container combo must first have all its containers well-formed. The last two conditions concern the relationships between the containers in the combo: each container must have the same number of elements if all abstractions are concretized; and for any concrete element x in a container Γ_j, there must exist an abstraction in each container Γ_k ($k \neq j$) that abstracts x.

Definition 6. *(*ADD, GET, and SWITCH*)* *An ADD(x) operation performed on a container combo* $\langle \Gamma_1, \ldots, \Gamma_{n-1}, i \rangle$ *adds a concrete element x to set* X_i *of the active container* Γ_i*, and appropriately updates abstract element set* A_j *in each inactive container* Γ_j *to add x's abstraction.*

A GET operation first looks for the target element in X_i *of the active container* Γ_i*. If the element does not exist in* X_i*, it finds the abstraction* $a \in \mathsf{A}_i$*, uses concretization function* γ_i *to retrieve all elements* $\gamma_i(a)$*, appropriately updates set* A_j *in each container* Γ_j *, and returns the target element* $x \in \gamma_i(a)$.

A SWITCH operation causes a container combo $\langle \Gamma_1, \ldots, \Gamma_{n-1}, i \rangle$ *to become* $\langle \Gamma_1, \ldots, \Gamma_{n-1}, j \rangle$ *($i \neq j$), without changing the internal state of each container.*

These definitions specify the behaviors of the three most important container operations. Section 4 discusses how these operations are performed for different types of containers such as `List`, `Map`, and `Set`.

Table 1. Containers currently supported by CoCo

Combo type	*Participating containers*	*Condition*	*Switch To*
java.util.List	ArrayList	#contains $>$ X $\wedge$ size $>$ Y	HashArrayList
	LinkedList	#get $>$ X $\wedge$ size $>$ Y	ArrayList
	HashArrayList	#contains $<$ X $\vee$ size $<$ Y	ArrayList
java.util.Map	HashMap	size $<$ Y	ArrayMap
	ArrayMap	size $>$ Y	HashMap
java.util.Set	HashSet	size $<$ Y	ArraySet
	ArraySet	size $>$ Y	HashSet

Definition 7. *(*Soundness of container switch*)* *A SWITCH operation that causes a container combo* C *to become* C' *is a sound switch if any element added by an ADD operation on* C *can be successfully retrieved by a corresponding GET operation on* C'*.*

Lemma 1. *(*Well-formedness implies soundness*)* *If the well-formedness of a container combo* C *is preserved by each (ADD or GET) operation, any SWITCH operation that occurs on* C *is sound.*

Proof Sketch. If C is always well-formed during the execution, an element added to C is either in the concrete element set X_i of the active container Γ_i, or in set X_j of an inactive container Γ_j and appropriately abstracted by an abstraction $a \in \mathsf{A}_i$. In the former case, it can be found directly in X_i, while in the latter case, it can be found by using γ_i to concretize abstraction a (as long as γ_i is sound). □

This section presents a methodology for creating sound container combos. Optimizations of any container implementations are guaranteed to be sound if they are instances of this formal framework. The library designers are responsible for creating appropriate optimizations for their containers and make them strictly follow the CoCo methodology. The next section describes the 7 containers CoCo currently supports and our implementations of their combos. We will also demonstrate, for each type of combo, how our specific implementations of abstraction and concretization preserve its well-formedness.

4 Implementation

We have modified four Java built-in container classes and implemented another three from scratch to instantiate the framework discussed in Section 3. Table 1 shows the container combos, when switches are performed, and switch destinations. Note that a subset of the switch conditions is adapted from the rules discovered and used in Chameleon [7]. Additional rules can be easily added by modifying the method `doProfiling` in each combo class (shown in Figure 2 (b)). Among the seven containers, `HashArrayList`, `ArrayMap`, and `ArraySet` are implemented by ourselves; the rest of them are modified from their original versions in the GNU classpath library [1].

These containers are selected because (1) they are all designed for general use, and (2) each container has clear time efficiency in a certain usage scenario. Future work may

[1] `www.gnu.org/s/classpath/`

Algorithm 1. List ADD that inserts object x at the index $index$ into a combo whose active container is i.

Input: List combo $\langle \Gamma_1, \ldots, \Gamma_{n-1}, i \rangle$, Index $index$, Element x
Suppose L_j is the underlying data structure in List Γ_j that stores X_j and A_j
foreach Γ_j, $j \in [1, n-1]$ **do**
  $newIndex \leftarrow$ recalculateIndex(L_j, $index$)
  if $j = i$ **then**
    /* update the active List */
    if $\mathsf{L}_j[newIndex]$ *is an abstract element* $a = \langle k, [index_m, index_n] \rangle$ **then**
      /* k is the ID of the List that contains the concrete element, $[index_m, index_n]$ is the predicate specifying a region*/
      split this abstraction into $a_1 = \langle k, [index_m, index - 1] \rangle$ and $a_2 = \langle k, [index + 1, index_n + 1] \rangle$
      replace a with $[a_1, x, a_2]$ in L_j
      foreach *abstract element* $\langle t, [index_q, index_r] \rangle$ *following* a_2 *in* L_j **do**
        update predicate $[index_q, index_r]$ to $[index_q + 1, index_r + 1]$
    else
      /*This is a concrete element*/
      insert x into L_j at the index $newIndex$
  else
    /*update an inactive List*/
    /*Initially each inactive List has one abstraction $\langle i, [0, 0] \rangle$ */
    if $\mathsf{L}_j[newIndex]$ *is an abstraction* $a = \langle k, [index_m, index_n] \rangle$ **then**
      if $k = i$ **then**
        update a to be $\langle k, [index_m, index_n + 1] \rangle$
      else
        create new abstraction $a' = \langle i, [index, index] \rangle$
        split a into $a_1 = \langle k, [index_m, index - 1] \rangle$ and $a_2 = \langle k, [index + 1, index_n + 1] \rangle$
        replace a with $[a_1, a', a_2]$ in L_j
        $a \leftarrow a_2$
    else
      /*$\mathsf{L}_j[newIndex]$ is a concrete element*/
      create new abstraction $a' = \langle i, [index, index] \rangle$
      insert a' into L_j at the index $newIndex$
      $a \leftarrow a'$
    foreach *abstract element* $\langle t, [index_q, index_r] \rangle$ *following* a *in* L_j **do**
      update predicate $[index_q, index_r]$ to $[index_q + 1, index_r + 1]$
return

Function **recalculateIndex**(L, $index$)
Input: data structure L, the original index given by the client $index$
Output: The new index in the presence of abstractions
$count \leftarrow 0$
foreach $k \in [0, |\mathsf{L}| - 1]$ **do**
  if $\mathsf{L}[k]$ *is an abstract element* $\langle t, [index_m, index_n] \rangle$ **then**
    if $index_m \leq index \leq index_n$ **then**
      return k
    $count \leftarrow count + (index_n - index_m + 1)$
  else
    if $count = index$ **then**
      return k
    $count \leftarrow count + 1$
return k

Algorithm 2. List GET that retrieves an element at the index $index$ from a combo whose active container is i.

Input: List combo $\langle \Gamma_1, \ldots, \Gamma_{n-1}, i \rangle$, Index $index$
Output: Element x

```
1  newIndex ← recalculateIndex(L_i, index)
2  if L[newIndex] is an abstract element a = ⟨k, [index_m, index_n]⟩ then
3      newIndex_m ← recalculateIndex(L_k, index_m)
4      newIndex_n ← recalculateIndex(L_k, index_n)
5      /*Concretize this abstraction to get the range of elements it represents*/ Array r ← getElements(L_k,
       newIndex_m, newIndex_n)
6      replace a with r in L_i
7      create abstraction a' = ⟨i, [index_m, index_n]⟩
8      replace r with a' in L_k
9      size ← index_n − index_m + 1
10     foreach Γ_j, j ∈ [1, n − 1] ∧ j ≠ i ∧ j ≠ k do
11         index_j ← recalculateIndex(L_j, index)
12         L_j[index_j] must be an abstraction a'' = ⟨k, [index_p, index_q]⟩
13         if index_q − index_p + 1 = size then
14             L_j[index_j] ← ⟨i, [index_p, index_q]⟩
15         else
16             split a'' into a_1 = ⟨k, [index_p, index_m − 1]⟩ and a_2 = ⟨k, [index_n + 1, index_q]⟩
17             create new abstraction aa = ⟨i, [index_m, index_n]⟩
18             replace a'' with [a_1, aa, a_2] in L_j
19     newIndex ← recalculateIndex(L_i, index)
20 return L[newIndex]
```

investigate container replacement based on other performance metrics (such as space efficiency and concurrency). In this section, we show only the ADD and GET operations for each combo, while our implementation supports all operations of the containers shown in Table 1. The implementations of the CoCo `Map`, `List`, and `Set` combos are publicly available at `www.ics.uci.edu/~guoqingx/tools/coco.jar`.

4.1 List Combo

The structure of the `List` combo has been shown in Figure 2. Here we discuss only how abstractions and concretizations are performed. Algorithm 1 illustrates the ADD operation of the `List` combo. As mentioned in Section 3, for each `List` in the combo, concrete elements and abstract elements are stored together in its underlying data structure that used to contain only concrete elements. Suppose L is such a data structure (e.g., the data array for `ArrayList`, the linked structure for `LinkedList`, etc.). We use $\mathsf{L}[k]$ to denote the k-th element in L, regardless of the type of L. Note that the sequence of elements in L is determined by the partial order $\prec$ of the container, which is, in turn, determined by the property encoding function f. For all `List` containers, f maps each element to a unique index, which is used to determine the position of this element in L.

For `List`, each abstraction has the form $\langle k, [m, n] \rangle$, where k is the ID of the container that has the concrete element and $[m, n]$ is the predicate that specifies a range of indices of the concrete elements represented by this abstraction. Note that CoCo currently allows only contiguous abstractions, and thus, a range is sufficient to represent a predicate. Richer predicates can be used if the contiguousness requirement is relaxed

in the future. Figure 5 has illustrated a simple (well-formed) `List` combo where both participating containers have abstractions.

Implementation of List ADD. Given an index $index$, we first compute a new index $newIndex$ that corresponds to $index$ in the presence of abstractions (line 3 in Algorithm 1). Function `recalculateIndex` is shown in lines 35–46. Next, we add this incoming object x into the active container Γ_i (lines 4–15). If the element at $newIndex$ is an abstract element (line 6), this abstraction is split into two separate abstractions (line 9) and then x is inserted between them (line 10). In addition, as the addition of x increases the index of each existing element following x by 1, we need to update the predicates in all the abstractions following a_2 in L to make them reflect the new indices (lines 11–12) (note that a_2 itself has been updated by line 9). In our implementation, abstractions in each L are indexed by a separate array, which allows quick update of the predicates.

Abstractions in all the inactive `Lists` need to be updated in a similar manner (lines 16–33) to account for this new element. In each inactive `List` Γ_j (where $j \neq i$), we first find the element at the index $newIndex$. If it is an abstraction and the host container in this abstraction happens to be the active `List` (line 20), we simply grow the range by 1 (line 21). Otherwise, we have to split this abstraction into two separate abstractions and insert a new abstraction a' (whose host container is the active container) between them (lines 23–25). If $\mathsf{L}_j[newIndex]$ is a concrete element, a new abstraction a' is created and inserted into L_j (lines 28-30). Eventually, the predicate in each abstraction following a is updated with a new index range (lines 32–33).

It is clear that if no switch is performed on the combo, the active `List` has all concrete elements and there is only one abstraction in each inactive `List` that abstracts all of them.

Implementation of List GET. Shown in Algorithm 2 is the GET operation implemented in CoCo. If the element at the index $newIndex$ is a concrete element, it is returned immediately (line 20). Otherwise, we retrieve all the concrete elements represented by the abstraction (lines 3–5) and bring them into the active container (line 6). An abstraction is then created to replace them in their original host container Γ_k (lines 7–8). As the host container of these elements is changed from Γ_k to Γ_i, code at lines 9–18 updates their corresponding abstractions in other inactive containers (not Γ_k or Γ_i) with the new host information. If the abstraction abstracts exactly the same range of concrete elements (line 13), we simply change its host container from k to i (line 14); otherwise, this abstraction needs to be split into two a_1 and a_2, and a new abstraction aa is created to represent this range (line 17). aa is inserted between a_1 and a_2 in L_j.

Note that once an abstraction a is encountered during a retrieval, we concretize the abstraction and move the entire range of concrete elements into the active container (lines 5–8). This is conceptually similar to a cache line fill. An alternative is to split a into three abstract elements $\langle k, [index_m, index]\rangle$, $\langle k, [index, index]\rangle$, and $\langle k, [index + 1, index_n]\rangle$, and then concretize only $\langle k, [index, index]\rangle$ to get the exact element requested. We have also implemented this approach for the general GET operation but found that it is much less effective than the one shown in Algorithm 2. It is

primarily because elements in a contiguous region are often visited together (e.g., using an Iterator) and getting them one at a time can increase cache misses significantly, especially for array-based containers.

However, we do use this alternative approach to implement methods that are specific to each container implementation (i.e., that are not declared in the common Interface). For example, method `getLast` (that is specific to `LinkedList`) needs to retrieve only the last element. If this last element in the `LinkedList` is a concrete element, it is directly returned; if it is an abstraction ($\langle k, [start, |\mathsf{L}| - 1]\rangle$), we first split it into two abstractions ($\langle k, [start, |\mathsf{L}| - 2]\rangle$, $\langle k, [|\mathsf{L}| - 1, |\mathsf{L}| - 1]\rangle$), and concretize only ($\langle k, [|\mathsf{L}| - 1, |\mathsf{L}| - 1]\rangle$) that represents the last element. The insight here is that in most cases where these implementation-specific methods are invoked, their containers are in the inactive state. We should avoid bringing many concrete elements from an active container back to an inactive one, which may lead to significant performance degradation.

Implementation of HashArrayList. Container `HashArrayList` is an `ArrayList`-based data structure. It maintains an `ArrayList` and a `HashMap` internally, and duplicates concrete elements between them. The `HashMap` contains only concrete elements and is used to perform `contains`-related operations. Abstractions are only allowed to be added into the `ArrayList`. When a `HashArrayList` is activated, all concrete elements in the combo are added into its `HashMap`. Abstraction/concretization operations are performed only between the "list" part of the `HashArrayList` and other lists in the combo; its "map" part always contains concrete elements as long as the `HashArrayList` is active.

Theorem 1. *(*List combo soundness*) Any SWITCH operation performed on the List combo (whose ADD and GET are shown in Algorithm 1 and Algorithm 2, respectively) is a sound switch.*

Proof Sketch. To prove this, it is important to show that each ADD and GET preserves the well-formedness of the combo. This can be easily seen from Algorithm 1 and Algorithm 2: (1) each element added to the active list (lines 4–15 in Algorithm 1) is well abstracted in each inactive list (lines 17–33 in Algorithm 1); and (2) each concretization replaces an abstract element in the active list with the concrete elements it represents (lines 5–6 in Algorithm 2). These concrete elements (in their old host) are replaced with a new abstract element (lines 7–8 in Algorithm 2). Abstractions corresponding to these elements in all the other inactive lists are updated to point to their new locations (lines 10–18 in Algorithm 2). Hence, well-formedness is preserved by both the element addition and the concretization. □

4.2 Map and Set Combos

The general algorithms (in Algorithm 1 and Algorithm 2) used for `List` can be naturally adapted to create `Map` and `Set` combos. For example, for a `Map` combo, when a pair of objects is added into the `HashMap` object (which is active), we can create an abstraction in the `ArrayMap` whose predicate records the bucket index of this pair in the `HashMap`. However, unlike the `List` combo where all participating `Lists` have

the same property encoding function f (thus we can easily merge adjacent abstractions as they represent adjacent concrete elements), `HashMap` and `ArrayMap` have different f. Elements in a `HashMap` are ordered based on their hashcodes while elements in an `ArrayMap` are ordered simply based on indices. As such, adjacent abstractions in the `ArrayMap` may represent concrete elements far away from each other in the `HashMap`. These abstractions may not be easily merged and we may end up with a great number of abstractions in each container, leading to increased space consumption and concretization overhead.

We develop a simpler approach for both `Map` and `Set` combos—for each combo, we maintain only one single abstraction in each inactive container that represents all concrete elements; upon a container switch, this abstraction is immediately concretized by moving the entire collection to the newly active container. We do not split abstractions because it may not be as beneficial for `Map` and `Set` as for `List`. Furthermore, as `ArraySet` and `ArrayMap` are designed to hold a very small number of elements, it is relatively inexpensive to perform MOVE_ALL upon each switch (e.g., much less costly than performing MOVE_ALL for a `List`).

Theorem 2. *(*Map and Set combo soundness*)* *Any SWITCH operation performed on the Map/Set combo is a sound switch.*

Proof Sketch. It is clear to see that the active container always has all the concrete elements. Each ADD or GET operation only updates the active container, and hence, the well-formedness of the combo is guaranteed. □

4.3 Discussion

Thread Safety. Our combo implementations (described in this section) are thread-safe. Because the client code always interfaces with methods in the original container, the concurrent behavior of the program is not influenced by any container switch. To illustrate, consider a switch from a `Hashtable` (which is thread-safe) to a `HashMap` (which is thread-unsafe). Because the client still invokes methods in `Hashtable` after the switch and these methods are appropriately synchronized, the fact that the actual service is provided by `HashMap` would not create any side effect. In addition, because the list of all associated containers is created inside the original container, whether or not these associated containers can be shared among threads depends on whether the original container is shared. No replacement will change the sharing property of the container (e.g., from being shared to thread-local or vice-versa). However, concurrent containers (e.g., those in `java.util.concurrent`) often use non-blocking algorithms (e.g., compare-and-swap). Forming a combo with both concurrent containers and non-concurrent, thread-unsafe containers may cause concurrency issues, which should not be allowed. Mixing only concurrent containers can be thread-safe as long as the abstraction and concretization operations are appropriately synchronized.

Handling of Operations with Incompatible Specifications. In some cases, implementations of the same operation (e.g., a method declared in a Java interface) in different containers may be incompatible, making it difficult for combo containers to provide the same service to the client. For example, different `Set` implementations may iterate

elements in different orders; unexpected program behaviors may result from switching implementations and traversing elements in a different order. The easiest way to solve the problem is to select only containers whose common methods have the same pre- and post-conditions to form a combo. However, this approach can limit significantly the optimization capabilities.

An alternative is to switch back to the original (user-chosen) container immediately upon the invocation of a method that is incompatible with its corresponding method in the original container. Concretizations are subsequently performed to bring necessary elements back. While this approach preserves semantics, it may potentially cause frequent container switches and element copies, leading to increased overhead. One possible way to alleviate the problem is to disable CoCo optimizations if such incompatible methods are frequently invoked. For example, the entire combo can be dropped; a detailed description of this optimization is described in Section 5. Note that such a case may rarely happen in practice. If a method is declared in an interface, its behaviors are often specified by the interface and all its implementations should strictly comply to this specification. Even if an implementation may have a unique way to fulfill the specification, this uniqueness should not be exploited by the client.

5 Optimizations

Naively profiling all container objects is expensive. This section describes three optimizations to reduce the replacement overhead. We modify Jikes Research Virtual Machine (RVM) to replace each allocation site of the form `java.util.X` x = new `java.util.X` (...) with a new allocation site of the form `coco.util.X` x = new `coco.util.X` (..., `null`), where `X` is a container class in Table 1. The additional argument `null` is used to notify the constructor that a new `XCombo` object needs to be created (e.g., line 8 in Figure 2 (a)). Note that such program modification can also be done by bytecode rewriting (e.g., through the load-time instrumentation framework `java.lang.instrument`). Implementing it inside a JVM eliminates the need to perform a separate program transformation phase.

5.1 Dropping Combos

A large part of the overhead comes from the method call forwarding and dispatch. For example, for each element addition/retrieval, there are two additional calls made (e.g., `LinkedList.get` calls `ListCombo.get`, which then calls `ArrayList.get$CoCo`). To reduce this overhead, we maintain a counter in each `XCombo` object, which is incremented every time method `doProfiling` (e.g., line 21 in Figure 2 (b)) is executed but no switch occurs. It is reset to zero if the combo decides to perform a switch. When this counter exceeds a pre-set threshold value, we no longer profile the operations—the currently active container appears to be suitable for the execution.

At this point, there are two situations that might occur. First, no replacement has ever been performed on the combo. The active container is the original container created by the client. In this case, the combo object notifies the active container to drop the combo (and all inactive containers) by setting the field `combo` to `null`. The container goes

back to normal and all operations afterwards will be performed directly on it. In the second case, switches have occurred and the currently active container is not the original (user-chosen) container. In this case, we cannot remove the combo because this current container does not directly communicate with the client. However, we can stop profiling the container operations to reduce overhead. Specifically, method `doProfiling` is no longer invoked for the future container operations.

While dropping combos is an important technique to reduce overhead, it may potentially lead to inappropriate switch decisions. For example, the usage of a container may change dramatically after its combo is dropped and the profiling is disabled, leaving the program with a suboptimal implementation. However, we have not found this problem affects the current implementations of the CoCo combos. In fact, many (long-lived) containers in a real-world program have strictly monotonic behaviors—they are more heavily used in a later stage of the execution (e.g., workload run) than in an earlier stage (e.g., warmup), making CoCo switch containers from an implementation that favors fewer elements/operations to another that favors more elements/operations.

5.2 Sampling

Invoking method `doProfiling` for each container operation can be quite expensive. A sampling-based profiling approach is employed to reduce this run-time cost. In addition, we do not profile a container until the first non-ADD operation is performed. This allows the container to have a start-up phase where it gets populated and stabilized; otherwise, the many ADD operations in the beginning may prevent CoCo from observing its real usage pattern and making appropriate switch decisions.

5.3 Lazy Creation of Inactive Containers

In the example shown in Figure 2, inactive containers are created immediately after the active container is created. However, if the combo never switches the container, it is completely redundant to create, initialize, and garbage collect these inactive containers. To make the implementation more efficient, we employ a lazy approach that does not allocate and initialize inactive containers until the first switch is about to be performed. This approach is sound, because, in any combo, each inactive container must have one single abstraction (that abstracts all existing concrete elements) before the first switch. We do not need to create and maintain this abstraction every time an element is added before a switch occurs.

6 Limitations

While the CoCo methodology is general enough to be applied to a variety of container implementations, it has the following three limitations. First, it improves application running time at the cost of introducing space overhead. While this overhead is relatively small (e.g., the detailed statistics are reported in Section 7) and acceptable for most applications (on machines with large memory space), the technique may not be suitable for optimizing memory-constrained programs.

The second limitation is that this technique does not preserve asymptotic complexity of the user-chosen containers. The containers used by CoCo at run time may have a different worst-case complexity than those intended to be used by the user and may thus perform worse in a later execution state when the CoCo profiling is disabled. However, as discussed earlier in this section, we found that the usage scenarios of most long-lived containers are quite consistent during the execution, which reduces the chance for CoCo to make inappropriate decisions. An approach similar to dynamic feedback [13] may be employed in the future work to run a program in a mixed profiling and production mode—the execution alternates between the profiling run and the production run so that profiling is periodically enabled to collect the most updated information.

Finally, although the framework presented in Section 3 provides soundness guarantee for combo implementations that comply with the CoCo methodology, there is no automated enforcement of this compliance. Library designers have to manually ensure that their combos are well-formed, implementations in each combo do not have conflicting specifications, and the replacement rules are appropriately placed. It is interesting to develop tool support, in the future, that can check the combo well-formedness to help developers write reliable optimizations.

7 Empirical Evaluation

We have evaluated CoCo on both a set of micro-benchmarks and a set of large-scale, real-world applications. All experiments are executed on a quad-core machine with an Intel Xeon E5620 2.40GHz processor, running Linux 2.6.18.

7.1 Micro-benchmarks

We have designed several micro-benchmarks (whose execution is dominated by container operations) in order to gain a deep understanding of what achieves efficiency and what incurs overhead. In fact, many of optimization techniques presented in Section 5 are motivated by our observations on the executions of these micro-benchmarks.

LinkedList $\rightarrow$ ***ArrayList***. This simple program creates 10 `LinkedList` objects. For each of them, 1000 elements are added and then 40,000 ADD/GET/REMOVE operations are performed. The original program finishes in 127.1 seconds. Using CoCo, its running time ranges from 35.2 to 38.8 seconds, depending on the threshold value X used to switch the `List`. Here X is the percentage of `GET` among all operations executed on each `LinkedList`. For this program, we have tried six different X (i.e., 0, 10%, ..., 50%) and found that the larger X is, the longer the running time is. `ArrayList` outperforms `LinkedList` in all kinds of operations (not just `get`(i)). Hence, we set X = 0 when we run experiments with large, real-world programs—`LinkedList` is immediately switched to `ArrayList` after the first $1/Y$ operations are performed on it (Y is the sampling rate).

ArrayList $\rightarrow$ ***HashArrayList***. We find that this switch is highly beneficial for programs with a large number of `contains` operations. We write a program that creates 100 `ArrayList` objects and populates each of them with 10,000 Integers. For each `ArrayList`, we generate 4,000 random Integers and test if this `List` contains these

Table 2. CoCo run-time statistics. Reported in section (a) and (b) is the run-time information of each program executed without and with CoCo, respectively; each section includes running time (T) in seconds, GC time (GC) in seconds, and peak memory consumption (S) in Megabytes; α in section (b) shows the standard deviation of the collected running times.

Bench	(a) Regular execution			(b) CoCo execution			
	$T_1(s)$	$GC_1(s)$	$S_1(Mb)$	$T_2(s)$	α (s)	$GC_2(s)$	$S_2(Mb)$
bloat	53.8	4.5	91.5	51.4(96%)	1.4 (2.7%)	5.9(131%)	165.2(181%)
chart	17.6	4.9	188.2	16.2(91%)	0.8 (4.9%)	7.4(151%)	228.2(121%)
fop	2.5	0.3	49.2	2.1(84%)	0.06 (2.9%)	0.4(133%)	50.1(102%)
lusearch	4.7	1.4	28.6	3.1(66%)	0.07 (2.2%)	1.4(100%)	28.6(100%)
avrora	23.6	1.5	62.6	20.8(89%)	1.1 (5.3%)	2.1(140%)	66.4(106%)
GeoMean				85.4%	3.4%	129.8%	118.8%

numbers. If an Integer is in the list, it is removed. This usage scenario of `ArrayList` is common in static analysis tools that make heavy use of worklist-based algorithms. The original running time is 149 seconds, while CoCo reduces it to 2.2 seconds (i.e., 74× speedup).

Set and Map Optimizations. The goal here is to investigate the borderline between `HashSet` (`HashMap`) and `ArraySet` (`ArrayMap`) (i.e., what is the appropriate size threshold under which `ArraySet`/`ArrayMap` would outperform `HashSet`/`HashMap`). We write a few programs that make heavy use of `Set`s and `Map`s, and use different size threshold values to tune the performance. It appears that, for `Set`, this line is somewhere between 5 and 8—in general, we observed, in this test, that switching from `HashSet` to `ArraySet` when the number of elements it contains is smaller than 5 is always beneficial. For `Map`, this line is lower—clear running time reduction can only be seen when we set this threshold to 2. This may be because `HashMap` operations are less expensive than those of `HashSet`, as `HashSet` maintains an internal `HashMap` and delegates all work to it. Based on these observations, we use 5 and 2 as size threshold values for switching `Map` and `Set` when running large benchmarks.

Note that even if small programs are used to tune the parameters, the usage of containers in these programs is real and each container has a large number of elements. In addition, it is much easier to see the impact of the parameter adjustment on performance in such container-centric programs than real-world programs whose performance can often be influenced by many complicated factors.

7.2 Performance on Large Benchmarks

Our large-scale benchmark set contains five real-world applications: bloat, chart, fop, lusearch, and avrora. These applications are chosen because container bloat has been previously found in them (e.g., reported in [7] and [3]). The sampling rate is 1/50, meaning that the method `doProfiling` is invoked once per 50 container operations. We have tried several different sampling rates and 1/50 appears to lead to the best performance. The generational Immix [14] garbage collector is used for our experiments.

For each program, we run it 10 times with a maximum 1GB heap (each with two iterations) using the large workload. The median steady-state performance and the standard

Table 3. Statistics of containers and their run-time switches

Bench	(a) CoCo containers				(b) #Switches			
	#AL	*#LL*	*#HM*	*#HS*	*LL* $\rightarrow$ *AL*	*AL* $\leftrightarrow$ *HL*	*HM* $\leftrightarrow$ *AM*	*HS* $\leftrightarrow$ *AS*
bloat	3322.4K	1158.0K	67.0K	370.7K	108	76/0	645/81	13860/241
chart	25.7K	0	14.5K	0	0	96/0	204/0	2/0
fop	87.4K	0	93	0	0	10/0	17/0	0/0
lusearch	374	0	259	128	0	0/0	101/0	0/0
avrora	50	414.3K	787	14	0	0/0	22/0	4/0

deviation of the running times (α) are reported in Table 2. In addition to the execution information, section (b) in Table 2 also includes ratios between the numbers in each column of section (b) and those in its corresponding column of section (a). Percentages shown in parentheses are either overheads (if $> 100\%$) or improvements (if $< 100\%$). Overall, CoCo reduces program running time by 14.6%, at the cost of introducing a 18.8% overhead in memory space. For all applications, the running time benefit gained from using suitable algorithms can successfully offset the overhead caused by creating (and garbage collecting) extra objects and making extra calls. It is worth investigating, in the future work, how to further reduce the overhead. For example, using object inlining (that inlines the combo and other inactive container objects) may reduce the number of extra objects created, leading to lower GC overhead and smaller space consumption. Note that our optimization techniques (discussed in Section 5) are effective: without them, the average performance improvement for these 5 applications is 6.34%. Due to space limitations, the detailed performance comparison (with and without these optimizations) is omitted.

Table 3 shows, for each program, the number of instances of each container type (i.e., `ArrayList`, `LinkedList`, `HashMap`, and `HashSet`) that CoCo attempts to optimize (section (a)) and the number of container switches that CoCo actually performs (section (b)). All kinds of switches except `LL` $\rightarrow$ `AL` are bi-directional. Each column for a bi-directional switch `X` $\leftrightarrow$ `Y` reports pairs of numbers a/b: a in each pair is the number of switches from `X` to `Y` and b is the number of switches from `Y` to `X`. Note that switches in both directions have occurred during the execution of bloat. In addition, we find that more than half of `LinkedLists` in bloat continue to be switched to `HashArrayLists` after becoming `ArrayLists`. This observation shows that for many data structure objects, there do not exist single optimal solutions throughout the execution. Optimal implementations change as the execution progresses and, therefore, an online adaptive system is highly necessary for removing container inefficiencies.

Despite the many container objects that CoCo attempts to optimize (e.g., shown in section (a) of Table 2), there is only a small number of them for which optimizations are actually possible. Our combo dropping technique appears to be effective—when no optimization opportunity can be found, the overhead incurred by CoCo is negligible. The current version of CoCo focuses only on Java built-in containers, leading to a fairly limited pool of candidates that can form combos. Larger performance gains may be achieved if the technique can be employed to optimize user-defined, application-specific data structures. Another interesting future direction is to optimize only a selected subset of containers that are highly likely to be inefficiently used. This can be done by focusing

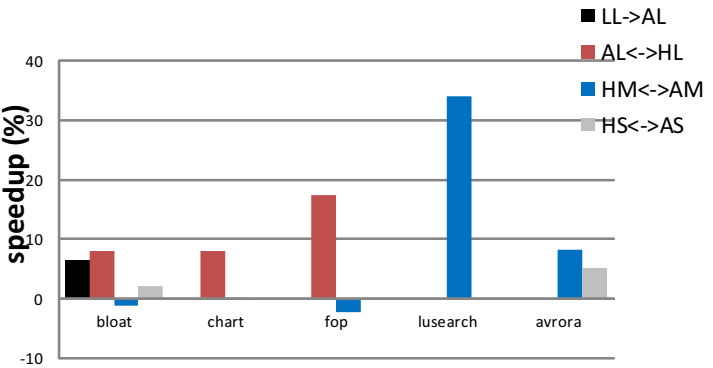


Fig. 6. Breakdown of the benefits from each type of container switch

on allocation sites. We may profile only a few instances (samples) of each allocation site and then use this information as feedback to guide the optimization of other instances created by the same allocation site.

7.3 Breakdown of the Benefits

To further understand the performance improvement that each type of switches contributes, an additional experiment is conducted: each application is run four times and for each time, only one type of switches (from Table 2) is enabled. The breakdown of the benefits is reported in Figure 6. In general, the effectiveness of each kind of replacement is application-specific and depends on the container usage in the application. However, it is clear that the switch from `ArrayList` to `HashArrayList` plays an important role in improving performance. In fact, we find an even larger benefit in fop (17.3%) when all the other types of switches are disabled. This is because `HashArrayList` has a clear algorithmic advantage for performing membership tests (e.g., `contains`) over other lists.

8 Related Work

Container Optimizations. Despite the body of work on container optimizations, CoCo is the first technique that can safely and automatically remove container inefficiencies.

Early work on the SETL framework [15,16,17] and recent work on data representation synthesis [18] attempt to generate appropriate data structure implementations from high-level abstractions at compile time. Our work addresses a different problem, which is to develop a system for Java that can safely switch implementations at run time. Recent attention has been paid to the container bloat problem [7,3,9]. Our previous work [3] proposes a static analysis to identify container inefficiencies. Work that is closest to our proposal is Chameleon [7] and Brainy [9]. Both of them can profile programs to make recommendations on appropriate container implementations that should be used. Recent work from [10] develops memory compaction techniques to reduce the footprint of Java collection objects. Our proposal differs from this category of research in

two major aspects. First, the problem that these tools address is orthogonal to the problem that we propose to solve. Both Chameleon and Brainy try to understand precisely what container implementation is suitable for what execution scenario (i.e., *when* to switch implementations), and then identify inconsistency between users' choices and the actual execution scenarios to help developers perform manual tuning. Our work goes far beyond and attempts to develop techniques that can automatically and safely switch implementations (i.e., *how* to switch implementations).

The C# standard library contains a collection class called `HybridDictionary` (`www.dotnetperls.com/hybriddictionary`) that implements a linked list and a hash table, switching over to the second from the first when the number of elements increases past a certain threshold. While this optimization is similar to our `ArrayList` $\rightarrow$ `HashArrayList` optimization, CoCo is a much more general technique that can be used to optimize a variety of containers.

Software Bloat Analysis. Software bloat analysis [19,4,20,1,21,22,3,2,23,24] attempts to identify and remove performance problems due to run-time inefficiencies in the code execution and the use of memory. Mitchell *et al.* [5] propose a manual approach that detects bloat by structuring behavior according to the flow of information, and their later work [4] introduces a way to find data structures that consume excessive amounts of memory. Our work takes further step in performing online optimization of inappropriately-used data structures. Shankar *et al.* [22] attempt to improve performance by making aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Our previous work [1,2] detects memory bloat by profiling copy chains and copy graphs, and by measuring costs and benefits of object-oriented data structures, respectively. Recent work [25] encodes runtime data structures to identify those that can be reused for improved efficiency.

All these existing techniques detect bloat and provide diagnostic report by profiling semantic information of the program execution. In this paper, we use one type of such information (i.e., container semantics) to find and remove problems automatically. Future work may identify additional semantic bloat patterns that can be exploited to perform similar semantics-aware optimizations.

9 Conclusions and Future Work

This paper proposes an application-level optimization technique, called CoCo, that can safely and adaptively switch container implementations. At the core of this technique is an abstraction-concretization methodology that can be used to create optimizable containers among which container replacement is guaranteed to be safe. While this work focuses on Java containers, the methodology can also be employed to optimize general user-defined data structures. Although the current implementation of CoCo suffers from a number of limitations, this work is the first step towards achieving the goal of automating a range of semantic optimizations for object-oriented applications, and our experimental results already show its promise. Future work may address these limitations and consider to extend the CoCo methodology to optimize languages like Scala, where some data structures seemingly expose their internal implementation through pattern matching. It is also interesting to investigate the possibility of offloading

expensive operations to idle cores (in a multicores architecture) to improve the replacement efficiency.

Acknowledgments. We would like to thank Per Larsen, Stefan Brunthaler, Dacong Yan, Brian Demsky, and the anonymous reviewers for their helpful comments.

References

1. 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)
2. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Schonberg, E., Sevitsky, G.: Finding low-utility data structures. In: PLDI, pp. 174–186 (2010)
3. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. In: PLDI, pp. 160–173 (2010)
4. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: OOPSLA, pp. 245–260 (2007)
5. Mitchell, N., Sevitsky, G., Srinivasan, H.: Modeling runtime behavior in framework-based applications. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 429–451. Springer, Heidelberg (2006)
6. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to Java runtime bloat. IEEE Software 27(1), 56–63 (2010)
7. Shacham, O., Vechev, M., Yahav, E.: Chameleon: Adaptive selection of collections. In: PLDI, pp. 408–418 (2009)
8. Xu, G., Rountev, A.: Precise memory leak detection for Java software using container profiling. In: ICSE, pp. 151–160 (2008)
9. Jung, C., Rus, S., Railing, B.P., Clark, N., Pande, S.: Brainy: effective selection of data structures. In: PLDI, pp. 86–97 (2011)
10. Gil, J., Shimron, Y.: Smaller footprint for java collections. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 356–382. Springer, Heidelberg (2012)
11. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: a language and compiler for algorithmic choice. In: PLDI, pp. 38–49 (2009)
12. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Edelman, A., Amarasinghe, S.: Language and compiler support for auto-tuning variable-accuracy algorithms, pp. 85–96 (2011)
13. Diniz, P.C., Rinard, M.C.: Dynamic feedback: an effective technique for adaptive computing. In: PLDI, pp. 71–84 (1997)
14. Blackburn, S.M., McKinley, K.S.: Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: PLDI, pp. 22–32 (2008)
15. Schonberg, E., Schwartz, J.T., Sharir, M.: Automatic data structure selection in SETL. In: POPL, pp. 197–210 (1979)
16. Schonberg, E., Schwartz, J.T., Sharir, M.: An automatic technique for selection of data representations in SETL programs. TOPLAS 3, 126–143 (1981)
17. Freudenberger, S.M., Schwartz, J.T.: Experience with the SETL optimizer. TOPLAS 5, 26–45 (1983)
18. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data representation synthesis. In: PLDI, pp. 38–49 (2011)
19. Mitchell, N.: The runtime structure of object ownership. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 74–98. Springer, Heidelberg (2006)
20. 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)

21. Mitchell, N., Schonberg, E., Sevitsky, G.: Making sense of large heaps. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 77–97. Springer, Heidelberg (2009)
22. Shankar, A., Arnold, M., Bodik, R.: JOLT: Lightweight dynamic analysis and removal of object churn. In: OOPSLA, pp. 127–142 (2008)
23. 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)
24. Chis, A.E., Mitchell, N., Schonberg, E., Sevitsky, G., O'Sullivan, P., Parsons, T., Murphy, J.: Patterns of memory inefficiency. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 383–407. Springer, Heidelberg (2011)
25. Xu, G.: Finding reusable data structures. In: OOPSLA, pp. 1017–1034 (2012)

Feature-Oriented Programming with Object Algebras

Bruno C.d.S. Oliveira[1], Tijs van der Storm[2], Alex Loh[3], and William R. Cook[3]

[1] National University of Singapore
oliveira@comp.nus.edu.sg
[2] Centrum Wiskunde & Informatica (CWI)
storm@cwi.nl
[3] University of Texas, Austin
{wcook,alexloh}@cs.utexas.edu

Abstract. Object algebras are a new programming technique that enables a simple solution to basic extensibility and modularity issues in programming languages. While object algebras excel at defining modular features, the *composition* mechanisms for object algebras (and features) are still cumbersome and limited in expressiveness. In this paper we leverage two well-studied type system features, *intersection types* and *type-constructor polymorphism*, to provide object algebras with expressive and practical composition mechanisms. Intersection types are used for defining expressive *run-time* composition operators (combinators) that produce objects with multiple (feature) interfaces. Type-constructor polymorphism enables generic interfaces for the various object algebra combinators. Such generic interfaces can be used as a type-safe front end for a generic implementation of the combinators based on reflection. Additionally, we also provide a modular mechanism to allow different forms of *self*-references in the presence of delegation-based combinators. The result is an *expressive*, *type-safe*, *dynamic*, *delegation*-based composition technique for object algebras, implemented in Scala, which effectively enables a form of *Feature-Oriented Programming* using object algebras.

1 Introduction

Feature-oriented programming (FOP) is a vision of programming in which individual *features* can be defined separately and then composed to build a wide variety of particular products [5,21,43]. In an object-oriented setting, FOP breaks classes and interfaces down into smaller units that relate to specific features. For example, the `IExp` interface below is a complete object interface, while `IEval` and `IPrint` represent interfaces for the specific features of evaluation and printing.

```
trait IExp {
  def eval() : Int
  def print() : String
}
```

```
trait IEval { def eval() : Int }

trait IPrint { def print() : String }
```

Existing object-oriented programming (OOP) languages make it difficult to support FOP. Traditionally OOP encourages the definition of complete interfaces such as `IExp`. Such interfaces are implemented by several classes. However adding

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 27–51, 2013.

a new feature usually involves coordinated changes in multiple classes. In other words, features often cut across traditional object-oriented modularity boundaries, which is centered on the behavior of individual objects. Such cross-cutting is a symptom of the *tyranny of the dominant decomposition* [48]: programming languages typically support development across one dominant dimension well, but all other dimensions are badly supported [23, 27, 48].

The main difficulty in supporting FOP in existing OOP languages stems from the intrinsic flexibility of FOP, which is challenging for programmers and language designers, especially when combined with a requirement for *modular type-checking* and *separate compilation*. Although research has produced many solutions to extensibility and modularity issues, most of these require advanced language features and/or careful advanced planning [11, 17, 19, 30, 32, 50, 52–54].

Object algebras [37] are a new approach to extensibility and modularity in OOP languages, which is based on a generalization of factories that creates families of related objects. The basic model of object algebras requires only simple generics, as in Java, without advanced typing features. For example, the following interface is an object algebra interface of simple expressions:

```
trait ExpAlg[E] {
  def Lit(x : Int) : E
  def Add(e1 : E, e2 : E) : E
}
```

Object algebras allow new features to be defined by implementing `ExpAlg`. For instance, classes implementing `ExpAlg[IPrint]` and `ExpAlg[IEval]` are algebras implementing printing and evaluation features respectively. Object algebras also allow extending the interface `ExpAlg` with new constructors [37]. As such object algebras provide a solution to the *expression problem* [14, 44, 51].

While object algebras excel at defining modular features, the *composition* mechanisms for object algebras (and features) are still cumbersome and limited in expressiveness. Combining algebras implementing `ExpAlg[IPrint]` and `ExpAlg[IEval]` to form `ExpAlg[IExp]` is possible, but tedious and cumbersome in Java. Moreover composition mechanisms must be defined separately for each object algebra interface, even though the composition follows a standard pattern. Finally, the basic model of object algebras does not support self-references, so overriding is not supported. The lack of good compositions mechanisms hinders the ability to express *feature interactions*, which is essential for FOP.

This paper provides object algebras with expressive and practical composition mechanisms using two well-studied type system features: *intersection types* [15] and *type-constructor polymorphism* [31, 45]. Both features (as well as their interaction) have been well-studied in programming language theory. For example Compagnoni and Pierce's $F^{\omega}_{\wedge}$ calculus [12], used to study language support for multiple inheritance, supports both features. Moreover, both features are available in the Scala programming language [33], which we use for presentation.

An intersection type, `A with B`, combines the interfaces `A` and `B` to form a new interface. Because the new interface is not required to have an explicit name, programmers can define generic interface composition operators, with types of the form `A => B => A with B`. These interface combinators allow object algebras to

be composed flexibly. While the interfaces are composed and checked statically, the composition of the algebras is done at runtime.

Type-constructor polymorphism refers to the ability for a generic definition to take a type constructor, or type function, as an argument. Since definitions like `ExpAlg` are type constructors, type constructor polymorphism is useful to abstract over such definitions. With type constructor polymorphism it is possible to define generic interfaces for object algebra combinators which are parametrized over the particular type of object algebras. In combination with meta-programming techniques this allows automating implementations of the combinators. As a result a single line is sufficient to implement the combinators for an extension or new object algebra interface. For example, the object `ExpComb`

```
object ExpComb extends Algebra[ExpAlg]
```

creates an object with combinators for the object algebra interface `ExpAlg`.

We also provide a modular mechanism to allow different forms of *self*-references in the presence of delegation-based combinators. As Ostermann [42] observes there are two important concerns related to self-references in delegation-based families of objects: 1) *virtual constructors*; 2) individual *object* self-references. The two issues are addressed using two types of self-references, which provide, respectively a notion of *family* and *object* self-references.

Ultimately, the object algebra composition mechanisms presented in this paper are *expressive*, *type-safe*[1],*dynamic* (composition happens at run-time), *delegation*-based and convenient to use. With these composition mechanisms a powerful and expressive form of FOP with object algebras is possible.

In summary, our contributions are:

- *FOP using object algebras:* We show that, provided with suitable composition mechanisms, object algebras enable a convenient and expressive form of FOP, which supports *separate compilation* and *modular type-checking*.
- *Generic object algebra combinators:* Using intersection types and type-constructor polymorphism, we show how to model general, expressive and type-safe composition mechanisms for object algebras.
- *Modular self-references:* We show a modular mechanism for dealing with self-references in the presence of delegation-based object algebra combinators.
- *Case studies:* We present two case studies that show the use of our techniques. The first is a typical test problem in FOP, the second involves composition and instrumentation of various operations on grammars. The code for the case studies and smaller examples, which has been successfully evaluated by the ECOOP artifact evaluation committee, is published online at:
  `https://github.com/tvdstorm/oalgcomp`

2 Object Algebras and Current Limitations

Object Algebras are classes that implement algebraic signatures encoded as parameterized interfaces, where the type parameter represents the carrier set of

[1] Uses of reflection are not *statically* type-safe, but they are optional and can be replaced by boilerplate type-safe code which avoids reflection.

```
trait ExpEval extends ExpAlg[IEval] {
  def Lit(x : Int) : IEval = new IEval {
    def eval() : Int = x
  }

  def Add(e1 : IEval, e2 : IEval) : IEval = new IEval {
    def eval() : Int = e1.eval() + e2.eval()
  }
}

object ExpEval extends ExpEval
```

Fig. 1. An object algebra for evaluation of integer expressions

```
trait ExpPrint extends ExpAlg[IPrint] {
  def Lit(x : Int) : IPrint = new IPrint {
    def print() : String = x.toString()
  }

  def Add(e1 : IPrint, e2 : IPrint) : IPrint = new IPrint {
    def print() : String = e1.print() + " + " + e2.print()
  }
}

object ExpPrint extends ExpPrint
```

Fig. 2. An object algebra for printing of integer expressions

the algebra [37]. In `ExpAlg` the methods `Lit` and `Add` represent the constructors of the abstract algebra, which create values of the algebra in the carrier type `E`. A class that implements such an interface is an *algebra* [22], in that it defines a concrete representation for the carrier set and concrete implementations of the methods. While it is possible to define an object algebra where the carrier set is instantiated to a primitive type, e.g. `int` for evaluation or `String` for printing, in this paper the carrier is always instantiated to an object interface that implements the desired behavior. For example, Fig. 1 and 2 define algebras for evaluating and printing expressions.

Provided with these definitions, clients can create values using the appropriate algebra to perform desired operations. For example:

```
def exp[E](f : ExpAlg[E]) : E =
   f.Add(f.Lit(5), f.Add(f.Lit(6),f.Lit(6)))

val o1 : IPrint = exp(ExpPrint)
val o2 : IEval = exp(ExpEval)
println("Expression: " + o1.print() + "\nEvaluates to: " + o2.eval())
```

defines a method `exp`, which uses the object algebra (factory) `f` to create values of an abstract type `E`. The example then creates objects `o1` and `o2` for printing and evaluation. The `ExpAlg` interface can be extended to define new constructors

for additional kinds of expressions. A new class that implements `ExpAlg` provides a new operation on expressions.

One serious problem with the example given above is that two versions of the object must be created: `o1` is used for printing, while `o2` is used for evaluation. A true feature-oriented approach would allow a single object to be created that supported both the printing and evaluation features. A more serious problem arises when one feature (or algebra) depends upon another algebra. Such operations cannot be implemented with the basic strategy described above. In general, feature *interactions* are not expressible.

The original object algebra proposal [37] addressed this problem by proposing *object algebra combinators*. Object algebra combinators allow the composition of algebras to form a new object algebra with the combined behavior and also new behavior related to the interaction of features. Unfortunately, the object algebras combinators written in Java lack expressiveness and are not very practical or convenient to use, for three different reasons:

- *Composed interfaces are awkward:* The Java combinators are based on creating *pairs* to represent the values created by combining two algebras. From the client's viewpoint, the result had the following form (using Scala, which has support for pairs):

  ```
  val o : (IEval,IPrint) = exp(combineExp(ExpEval,ExpPrint))
  println("Eval: " + o._1.eval() + "\nPrint: " + o._2.print())
  ```

  The value `o` does combine printing and evaluation, but such pairs are cumbersome to work with, requiring extraction functions to access the methods and revealing that the objects result from compositions. Combinations of more than two features require nested pairs with nested projections, adding to the usability problems.
- *Combinators must be defined for each object algebra interface:* There is a lot of boilerplate code involved because combinators must be implemented or adapted for each new object algebra interface or extension. Clearly, this is quite inconvenient. It would be much more practical if the combinators were automatically defined for each new object algebra interface or extension.
- *The model of dynamic composition lacks support for self-references:* Finally, combinators are defined using dynamic invocation, rather than inheritance. The Java form of object algebras does not support self-reference or *delegation*. Since self-reference is important to achieve extensibility, the existing object algebra approach lacks expressiveness.

As a result, while object algebras provide a simple solution to basic modularity and extensibility issues, existing composition mechanisms impose high overhead and have limited expressiveness for FOP. The remainder of the paper shows solutions to the three problems.

3 Combining Object Algebras with Intersection Types

Intersection types help with providing a solution to the problem of combining object algebras conveniently. Combining object algebras allows two different

behaviors or operations, implemented by two specific algebras, to be available at once. Intersection types avoid the heavy encoding using pairs and allow methods to be called in the normal way.

3.1 Scala's Intersection Types

In Scala, an intersection type[2] `A with B` expresses a type that has both the methods of type `A` and the methods of type `B`. This is similar to

```
interface AwithB extends A, B {}
```

in a language like Java or C#. The main difference is that an intersection type does not require a new nominal type `AwithB`. Furthermore, Scala's intersection types can be used even when `A` and `B` are type parameters instead of concrete types. For example,

```
trait Lifter[A,B] {
   def lift(x : A, y : B ) : A with B
}
```

is a **trait** that contains a method `lift` which takes two objects as parameters and returns an object whose type is the intersection of the two argument types. Note that such an interface cannot be expressed in a language like Java because it is not possible to create a new type that expresses the combination of *type parameters* `A` and `B`[3].

3.2 Merging Algebras Using Intersection Types

Intersection types allow easy merging of the behaviors created by object algebras. The `lift` operation defined in the previous section for combining objects is used in the definition of a `merge` operator for algebras. Conceptually, a `merge` function for an algebra interface F combines two F-algebras to create a combined algebra:

`merge`F`: (A => B => A with B) =>` F`[A] =>` F`[B] =>` F`[A with B]`

Unlike the solution with pairs described in Section 2, intersection types do not require additional projections. The additional function argument represents the `lift` function, of type `A => B => A with B`, that specifies how to compose two objects of type `A` and `B` into an object of type `A with B`. This lift function resolves conflicts between the behaviors in `A` and `B` by appropriately invoking (delegating) behaviors in `A with B` to either `A` or `B`. The lift function can also resolve *interactions* between features. In other words, the function argument plays a role similar to *lifters* in Prehofer's FOP approach [43].

From a conceptual point of view, the key difference between `combine` on pairs and `merge` is that the former uses a `zip`-like operation with pairs, and the latter uses a `zipWith`-like operation with intersection types.

[2] In Scala these are often called *compound types*.

[3] Note that Java supports a limited form of intersection types in generic bounds, but this form of intersection types is too weak for our purposes. In Java it is possible to have generic bounds such as `<T extends A & B>`, where `A & B` denotes an intersection type. However `A` and `B` cannot be type parameters: they must be concrete types.

```
trait ExpMerge[A,B] extends ExpAlg[A with B] {
  val lifter : Lifter[A,B]
  val alg1 : ExpAlg[A]
  val alg2 : ExpAlg[B]

  def Lit(x : Int) : A with B =
    lifter.lift(alg1.Lit(x),alg2.Lit(x))

  def Add(e1 : A with B, e2 : A with B) : A with B =
    lifter.lift(alg1.Add(e1, e2),alg2.Add(e1, e2))
}
```

Fig. 3. Expression merging combinator with intersection types

```
object LiftEP extends Lifter[IEval,IPrint] {
  def lift(x : IEval, y : IPrint) = new IEval with IPrint {
    def print() = y.print()
    def eval() = x.eval()
  }
}

object ExpPrintEval extends ExpMerge[IEval,IPrint] {
  val alg1 = ExpEval
  val alg2 = ExpPrint
  val lifter = LiftEP
}

def test2() = {
  val o = exp(ExpPrintEval)
  println("Eval: " + o.eval() + "\nPrint: " + o.print())
}
```

Fig. 4. Merging the printing and evaluation algebras

Figure 3 defines the `merge` combinator for expressions in Scala as the trait `ExpMerge`. The value of type `Lifter[A,B]` plays the role of the combination function in `merge`, while the two values `alg1` and `alg2` are the two object algebra arguments. The definition of `Lit` and `Add` uses the method `lifter` to combine the two corresponding objects, which are delegated by invoking the corresponding method on the arguments. Intersection types automatically allow the following subtyping relationships:

`A with B <: A` *and* `A with B <: B`

These relationships ensure that no conversion/extraction is needed when delegating arguments, for example, `e1` and `e2` in `Add`. This is an advantage over using pairs, because extraction of the arguments from the pairs is not needed.

Figure 4 illustrates how to merge the printing and evaluation algebras to create an `ExpPrintEval` algebra. Clients can use this factory to create objects of type `IEval with IPrint`, which include `print` and `eval` in a single interface. The result is a seamless combination of the printing and evaluation features.

```
class LiftDecorate[A](wrap : A => A) extends Lifter[A,Any] {
  def lift(x : A, y : Any) = wrap(x)
}

trait ExpDecorate[A] extends ExpMerge[A,Any] {
  val alg2 = ExpEmpty
  val lifter = new LiftDecorate(wrap)
  def wrap(x : A) : A
}
```

Fig. 5. Decoration combinator derived from merge and empty

Unfortunately, it is still necessary to define some boilerplate. Each merge requires a lifter object (`LiftEP`), which in this case combines `IEval` with `IPrint`. Often, as in this case, such lifting operations simply create the new object and delegate the methods to the corresponding methods in either `x` or `y`. As we shall see in Section 5.2, in such cases, it is possible to define a generic lifting behavior.

4 Applying Uniform Transformations to Object Algebras

This section shows how to define uniform transformations on object algebras using the merge combinator. This serves as a representative example of the expressive power of merge. The primary example of a uniform transformation is adding a generic tracing behavior at each step of evaluation. In this case the behavior being added is not specific to each constructor in the algebra, but is instead a uniform behavior at each evaluation step.

Uniform transformations are formalized as a combinator, which encapsulates a Decorator [20] wrapping each value constructed by the algebra with additional functionality. The `decorate` combinator takes a function `A =>A` and an object algebra `ExpAlg[A]` and produces a wrapped object algebra of type `ExpAlg[A]`:

```
decorate : (A => A) => ExpAlg[A] => ExpAlg[A]
```

Although `decorate` can be implemented directly, we choose to implement it in terms of the more generic `merge` combinator. In this use of `merge` it is the `lifting` function that matters, while the possibility to combine two algebras is not needed. As a result, we supply an `empty` algebra as the second algebra. Conceptually, the `decorate` combinator defines a `lifting` that applies the transformation `wrap` to its first argument, and ignores the second (empty) argument.

```
decorate wrap alg = merge(x => y => wrap(x), alg, empty)
```

Figure 5 gives the Scala definition of the `decorate` combinator for the expressions algebra. The `ExpDecorate` trait extends `ExpMerge` and sets the second algebra to an empty object algebra. An abstract method `wrap` specifies a decoration function, which is applied to objects of type `A`.

An empty algebra, defined in Fig. 6, is an algebra, of type `ExpAlg[Any]`, that does not define any operations. It instantiates the carrier type to `Any`, a Scala

```
trait ExpEmpty extends ExpAlg[Any] {
  def Lit(x : Int) : Any = new Object()
  def Add(e1 : Any, e2 : Any) : Any = new Object ()
}
object ExpEmpty extends ExpEmpty
```

Fig. 6. The `Empty` expression algebra

```
object TraceEval extends ExpDecorate[IEval] {
  val alg1 = ExpEval
  def wrap(o : IEval) = new IEval() {
    def eval() = {
      println("Entering eval()!")
      o.eval()
    }
  }
}
```

Fig. 7. A decorator for tracing

type that plays the role of the top of the subtyping hierarchy. Every method in the object algebra has the same definition: `new Object()`.

4.1 Tracing by Decoration

Figure 7 defines a tracing mechanism using the decorator combinator. The object `TraceEval` wraps an existing evaluator with tracing. The `eval` method first prints a message and then delegates to the base evaluator `o`. By extending `ExpDecorate[IEval]`, this wrapper is applied to every evaluator created by the underlying evaluator `ExpEval`. When `exp` is invoked with `TraceEval`:

```
val o : IEval = exp(TraceEval)
println("Eval: " + o.eval())
```

the string `Entering eval()!` is printed 5 times in the console.

5 Generic Object Algebra Combinators

To avoid manually writing boilerplate code for combinators such as `merge`, `empty` or `decorate`, we develop object algebra combinators interfaces and corresponding implementations generically.

A generic `merge` combinator defined on an object algebra interface F containing methods $m_1(args_1), ..., m_n(args_n)$ might look as follows:

```
trait MergeF[A,B] extends F[A with B] {
  val lifter : Lifter[A, B]
  val a1 : F[A]
  val a2 : F[B]
  def m_i(args_i) : A with B = lifter.lift(a1.m_i(args_i), a2.m_i(args_i))
}
```

```
trait Algebra[F[_]] {
  // Basic combinators
  def merge[A,B](mix : Lifter[A,B], a1 : F[A], a2 : F[B]) : F[A with B]
  def empty() : F[Any]

  // Derived combinator(s)
  def decorate[A](parent : F[A], wrap : A => A) : F[A] =
    merge[A,Any](new LiftDecorate(wrap),parent,empty)
}
```

Fig. 8. The generic interface for object algebra combinators

The generic `merge` combinator extends an interface which is polymorphic in the types of the algebras it combines. Its methods are thin facades to invoke the underlying lifter on the two object algebras. There are two challenges in defining a generic `merge`: the first is defining its generic interface and the second is implementing the constructors. The former is solved using *type-constructor polymorphism* [31, 45], and the latter with reflection. The same idea is applied to other combinators, including `empty`.

5.1 A Generic Interface for Object Algebra Combinators

Scala supports type-constructor polymorphism by allowing a trait to be parameterized by a generic type, also known as a type constructor. With type-constructor polymorphism it is possible to provide a generic interface for object algebra combinators, as shown in Fig. 8. The `Algebra` trait is parameterized by a type constructor `F[_]`, which abstracts over object algebra interfaces like `ExpAlg`. Note that the annotation `[_]` expresses that `F` takes one type argument. The trait contains three methods. These methods provide a generalized interface for the object algebra combinators introduced in Sections 3 and 4, using the type constructor `F` instead of a concrete object algebra interface.

The `Algebra` interface is inspired by *applicative functors* [29]: an abstract interface, widely used in the Haskell language, to model a general form of effects. In Haskell[4], the interface for applicative functors is defined as:

```
class Applicative f where
  merge :: (a → b → c) → f a → f b → f c
  empty :: f ()
```

Like object algebra combinators, applicative functors are also closely related to `zip`-like operations. Our combinators can be viewed as an adaptation of the applicative functors interface. However, an important difference is that applicative functors require co-variant type-constructors (the parameter type occurs in positive positions only), whereas object algebra interfaces do not have such restriction. In fact most object algebras use invariant type-constructors (the parameter type can occur both in positive and negative positions). To compensate

[4] The actual interface in the Haskell libraries is different, but equivalent in expressiveness to the one described here as discussed by McBride and Paterson [29].

```
def merge[A,B](lifter : Lifter[A,B], a1 : F[A], a2 : F[B])(implicit m :
    ClassTag[F[A with B]]) : F[A with B] =
  createInstance(new InvocationHandler() {
  def invoke(proxy : Object, method : Method, args : Array[Object]) = {
    val a = method.invoke(a1,args : _*)
    val b = method.invoke(a2,args : _*)
    lifter.lift(a.asInstanceOf[A],b.asInstanceOf[B]).asInstanceOf[Object]
  }})

def empty(implicit m : ClassTag[F[Any]]) : F[Any] =
  createInstance(new InvocationHandler() {
  def invoke(proxy : Object, method : Method, args : Array[Object]) =
      new Object()
  })
```

Fig. 9. Reflective, generic implementations of merge and empty

for the extra generality of the object algebra interface type-constructors, the `merge` operation restricts the type `c` to the intersection type `A with B`.

Finally note that `Algebra` can itself be viewed as an object algebra interface whose argument type (constructor) abstracts over another object algebra interface. `Algebra` can be thought of as a factory of factories, or a *higher-order* factory: the factory methods `merge` and `empty` construct factory objects.

Combinators for Specific Object Algebras. Using `Algebra` we can create an object that contains object algebra combinators specialized for `ExpAlg` as follows:

```
object ExpComb extends Algebra[ExpAlg] {
  def merge[A,B](f : Lifter[A,B], a1 : ExpAlg[A], a2 : ExpAlg[B]) =
    new ExpMerge[A,B]() {
      val lifter : Lifter[A,B] = f
      val alg1 : ExpAlg[A] = a1
      val alg2 : ExpAlg[B] = a2}
  def empty() = ExpEmpty
}
```

and use these combinators in client code. For example:

```
import ExpComb._
val o = exp(merge(LiftEP,decorate(ExpEval,TraceEval.wrap),ExpPrint))
println("Eval: " + o.eval() + "\n PP: " + o.print())
```

creates an object `o`, combining a traced version of evaluation and printing.

5.2 Default Combinator Implementations Using Reflection

Reflection can be used to implement `merge`, `empty`, and a default `Lifter` for any given object algebra interface. Reflection does not guarantee static type safety, but if the default reflective implementation is trusted, then the strongly-typed generic interface guarantees the static type-safety of the client code.

Figure 9 gives generic implementations for `merge` and `empty` using reflection. These implementations can be used in the generic interface `Algebra` to provide

a default implementation of the combinators. The idea is to use *dynamic proxies* [26] to dynamically define behavior. A dynamic proxy is an object that implements a list of interfaces specified at runtime. As such, that object can respond to all methods of the implemented interfaces by implementing the method `invoke` and the `InvocationHandler` interface. The dynamic proxy objects are created using the `createInstance` method:

```
def createInstance[A](ih : InvocationHandler)(implicit m : ClassTag[A]) = {
    newProxyInstance(m.runtimeClass.getClassLoader,
        Array(m.runtimeClass),ih).asInstanceOf[A]
}
```

This method relies on the JDK reflection API, which supports the creation of dynamic proxies, and Scala's *mirror-based* [8] reflection API to provide reflective information of type parameters. The use of Scala's mirror-based reflection requires an adjustment on the types of the combinators in `Algebra`. The combinators now need to take an additional *implicit parameter* [40] `m`, which contains a reflective description of type parameters. This additional parameter does not affect client code since it is implicitly inferred and passed by the Scala compiler.

Unfortunately there is a wrinkle in our use of reflection: while supported by Scala, intersection types are not natively supported by the JVM. As a result the use of `createInstance` to dynamically generate an object with an intersection type is problematic. Fortunately there is a workaround which consists of creating a *nominal subtype* `S` of an intersection type `A with B`. This allows working around the JVM limitations, but requires an extra type argument `S <: A with B` in our combinators. In the paper, for clarity of presentation, we will ignore this issue and assume that `createInstance` works well with intersection types.

Generic Lifters. The `delegate` function creates a generic lifter function.

```
def delegate[A,B](x : A, y : B)(implicit m : ClassTag[A with B]) =
  createInstance[A with B](new InvocationHandler() {
    def invoke(proxy : Object, method : Method, args : Array[Object]) = {
      try {
        method.invoke(x, args : _*)
      } catch {
        case e : IllegalArgumentException => method.invoke(y, args : _*)
      }}})
```

This function is quite useful to handle intersection types composed of interfaces whose methods are disjoint. An example is the intersection of `IEval` and `IPrint`. In the case that the sets of methods are not disjoint, methods in algebra `x` will have priority over those in algebra `y`.

With `delegate` and `merge` it is possible to define a combinator `combine`, which resembles the `zip`-like combinator with the same name proposed by Oliveira and Cook [37]. The difference is the result is an intersection type instead of a pair.

```
def combine[A,B](alg1 : F[A], alg2 : F[B])(implicit m1 : ClassTag[F[A
    with B]], m2 : ClassTag[A with B]) : F[A with B] =
  merge[A,B](new Lifter[A,B]() {
     def lift(x : A, y : B) = delegate[A,B](x,y)},
    alg1, alg2)
```

With `combine` we can combine features without having to explicitly define a lifter function. For example:

```
val o = exp(combine(decorate(ExpEval,TraceEval.wrap),ExpPrint))
println("Eval: " + o.eval() + "\n PP: " + o.print())
```

combines evaluation with printing and also enables tracing on `eval`.

In summary, generic object algebra combinators avoid manual definitions such as `ExpMerge` and `LiftEP`. Instead:

```
object ExpComb extends Algebra[ExpAlg]
```

creates a set of combinators, including `merge`, `empty`, `delegate`, `combine`, and `decorate` for `ExpAlg`, providing the necessary composition infrastructure. Nevertheless programmers can still provide their own definitions if desired, which can be useful to avoid performance penalties due to the use of reflection.

6 Object Algebras, Self-references and Delegation

This section defines a generalization of object algebra interfaces which accounts for *self*-references in our delegation-based setting. Self-references are orthogonal and complementary to the generic and reflective combinators presented in Section 5. As such, for simplicity of presentation, we will first present the treatment of self-references on a specific object algebra interface and then discuss the adaptations needed to the generic object algebra interfaces.

6.1 Generalizing Object Algebras to Account for *Self*-references

Since the programming style in this paper is based on delegation an important question is how to account for self-references. The standard self-references provided by Scala's built-in class-based inheritance model do not provide an adequate semantics in the presence of delegation (or run-time inheritance). As Ostermann [42] points out, when dealing with delegation-based object families there two important issues that need special care:

- *Object self-references:* When composing object algebras using combinators like `merge` or, more generally, delegating on another algebra, the self-reference to the constructed objects should refer to the *whole* composition rather than the individual object.
- *Virtual constructors:* Similarly to the semantics of *virtual classes* [19], the constructors of objects (that is the methods of the object algebras) should be late bound, and refer to the composite object algebras rather than the object algebra being defined.

Both of these problems can be solved using two types of self-references: *object* self-references and *family* self references. In order to account for these two types of self-references we first need a generalization of object algebra interfaces, as shown in Fig. 10. This generalization form has been studied before in the context

```
trait GExpAlg[In,Out] {
  def Lit(x : Int) : Out
  def Add(e1 : In, e2 : In) : Out
}

type ExpAlg[E] = GExpAlg[E,E]
```

Fig. 10. A generalized object algebra interface for expressions

```
trait ExpPrint2[S <: IEval with IPrint] extends OExpAlg[S, IPrint] {
  def Lit(x : Int) = self => new IPrint() {
    def print() = x.toString()
  }

  def Add(e1 : S, e2 : S) = self => new IPrint() {
    def print() = e1.print() + " + " + e2.print() + " = " + self.eval()
  }
}
```

Fig. 11. An object algebra with a dependency

of research on the relationship between the VISITOR pattern and Church encodings [41]. The idea is to distinguish between the uses of carrier types with respect to whether they are inputs (`In`) or outputs (`Out`). In type-theoretic terms, this means distinguishing between the positive and negative occurrences of the carrier type. It is easy to recover the conventional object algebra interface `ExpAlg[A]` simply by making the two type parameters in `GExpAlg` be the same.

6.2 Object Self-references

The generalized interface allows us to account for the type of object algebra interfaces, `OExpAlg`, with unbound (or open) object self-references:

```
type OExpAlg[S <: E, E] = GExpAlg[S, Open[S,E]]
type Open[S <: E, E] = (=> S) => E
```

The type `OExpAlg` is parameterized by two types `E` and `S`. The type `E` is the usual carrier type for object algebras. The type `S` is the type of the entire composition of objects, which must be a subtype of `E`. In `OExpAlg` the outputs are a function `(=>S)=> E`. The argument of this function (`=>S`) is the unbound self-reference, which is used by the function to produce an object of type `E`. To prevent early evaluation of the self argument, it is marked as a call-by-name parameter by placing `=>` before the argument type. Scala wraps call-by-name arguments in thunks to delay their evaluation until the function body needs their value.

Dependent Features. An example where using self references is important is when defining a feature which depends on the availability of another feature. Figure 11 illustrates one such case: a variant of the printing feature, which uses evaluation in its definition. The dependency on evaluation is expressed by bounding the

```
trait CloseAlg[E] extends ExpAlg[E] {
  val alg : OExpAlg[E,E]

  def Lit(x : Int) : E = fix(alg.Lit(x))
  def Add(e1 : E, e2 : E) : E = fix(alg.Add(e1,e2))
}

def closeAlg[E](a : OExpAlg[E,E]) : ExpAlg[E] = new CloseAlg[E] {
  val alg = a
}
```

Fig. 12. Closing object self-references

type of the self-reference `S` to `IEval with IPrint`. This imposes a requirement that the self-reference and the input arguments of the different cases (such as `e1` and `e2`) implement both evaluation and printing. However, `ExpPrint2` does not have any hard dependency on a particular implementation of `IEval` and it only defines the behaviour of the printing feature, though it may call evaluation in its implementation. The definition of the `print` method for `Add` uses the self reference (`self`). Note that Scala's built-in self-reference `this` is useless in this situation: `this` would have the type `IPrint`, but what is needed is a self-reference with type `S <: IEval with IPrint` (the type of the composition).

Closing Object Self-References. Before we can use object algebras with object self-references we must close (or bind) those references. This can be done using a closing object algebra, which is shown in Fig. 12. The closing algebra `CloseAlg[E]` extends the standard object algebra interface `ExpAlg` and delegates on an open object algebra `alg`, which is the algebra to be closed. In each case self-references are closed using *lazy fixpoints*. Lazy fixpoints are a standard way to express the semantics of *dynamic mixin inheritance* and bind self-references in denotational semantics [13] and lazy languages [35].

```
def fix[A](f : Open[A,A]) : A = {lazy val s : A = f(s); s}
```

To implement the lazy fixpoint we exploit Scala's support for lazy values. It is possible to achieve the same effect using mutable references, but Scala's lazy values provide a more elegant solution.

6.3 Family Self-references

Another interesting type of self-references are *family* self references. The type `OpenExpAlg` is the type of (open) object algebra interfaces with both object and family self-references[5]:

```
type OpenExpAlg[S <: E, E] = (=> ExpAlg[S]) => GExpAlg[S, Open[S,E]]
```

[5] Note that it is also possible to define a simpler type `(=>ExpAlg[S])=>ExpAlg[S]` which accounts only for object algebra interfaces with family self references.

```
trait ExpPrint3[S <: IEval with IPrint] extends SelfExpAlg[S,IPrint]{
  def Lit(x : Int) = self => new IPrint() {def print() = x.toString()}

  def Add(e1 : S, e2 : S) = self => new IPrint() {
    def print() = {
      val plus54 = fself.Add(fself.Lit(5), fself.Lit(4));
      e1.print() + " + " + e2.print() + " = " + self.eval() +
        "  and " + "5 + 4 = " + plus54.eval();
    }
  }
}

def ExpPrint3[S <: IEval with IPrint] : OpenExpAlg[S,IPrint] =
  s => new ExpPrint3[S] {lazy val fself = s}
```

Fig. 13. A printing operation using family and object self-references

Similarly to `OExpAlg`, `OpenExpAlg` is parameterized by two type parameters `E` and `S <: E`. `OpenExpAlg` is a function type in which the argument of that function (`=>ExpAlg[S]`) is the unbound family self reference. The output of the function is `GExpAlg[S, Open[S,E]]`. This type denotes that the input arguments in the algebra are values of the composition type `S`, whereas the output types have unbound object self references (just as in Section 6.2).

It is possible to define a generic interface for object algebras interfaces with family self references:

```
trait SelfAlg[Self <: Exp, Exp] {
   val fself : ExpAlg[Self]
}
```

This interface can be combined with particular object algebra interfaces to add family self references. For example:

```
trait SelfExpAlg[Self <: Exp, Exp] extends
  GExpAlg[Self,Open[Self,Exp]] with SelfAlg[Self,Exp]
```

denotes integer expression object algebra interfaces with family and object self references. As shown in Fig. 13, this interface can be used to define object algebras with both types of self references. The `ExpPrint3` object algebra implements a modified version of `ExpPrint2` which adds some additional behavior to the printing operation in the `Add` case. The idea is to extend `ExpPrint3` so that the algebra constructs a value denoting $5 + 4$ in one of the operations. The family self reference `fself` (which is available as a value from the extended interface `SelfAlg`) ensures that the constructors refer to the overall composed object algebra instead of the local `ExpPrint3` object algebra. It is the use of the family self-reference that enables a *virtual* constructor semantics. If `this.Add(this.Lit(5),this.Lit(4))` was used instead then the constructors would not have the expected semantics in the presence of compositions around `ExpPrint3`.

Closing References. Both object self references and family self references can be closed with the following definition:

```
trait Algebra[F[_,_]] {
  type FOpen[F[_,_],S <: T, T] = (=> F[S,S]) => F[S,Open[S,T]]

  // Basic combinators
  def merge[A,B,S <: A with B](mix : Lifter[A,B,S], a1 : FOpen[F,S,A], a2
      : FOpen[F,S,B]) : FOpen[F,S,A with B]
  def empty[S] : FOpen[F,S,Any]

  // Derived combinator(s)
  def decorate[A, S <: A](parent : FOpen[F,S,A], wrap : A => A) :
      FOpen[F,S,A] =
    merge[A,Any,S](new LiftDecorate(wrap),parent,empty[S])

  // closing combinators
  def fcloseAlg[S](a : F[S,Open[S,S]]) : F[S,S]
  def fclose[S](f : FOpen[F,S,S]) : F[S,S]
}
```

Fig. 14. Interface for generic combinators with self-references

```
def close[S](f : OpenExpAlg[S,S]) : ExpAlg[S] = fix(compose(closeAlg,f))
```

Essentially, `close` first binds the object self references using `closeAlg` and then it binds the family self-references using a lazy fixpoint. Note that `compose` is the standard function composition operation.

6.4 Generic Combinators with Self-references

The generic combinators presented in Fig. 8 can be adapted to account for self-references, as shown in Fig. 14. The trait `Algebra` now has to abstract over a type constructor with 2 arguments, to account for the generalized form of object algebra interfaces. The type `FOpen` is a generalization of `OpenExpAlg`, for some algebra `F` instead of the specific `ExpAlg`. The combinators `merge` and `empty` must work on open object algebras instead of closed ones. Moreover, in the `merge` combinator the `Lifter` trait needs to be updated slightly to allow the use of object self-references by the lifting functions:

```
trait Lifter[A,B, S <: A with B] {
  def lift(x : A, y : B) : Open[S, A with B]
}
```

Finally, generic forms of closing operators are included in the `Algebra` interface.

As with the combinators in Fig. 8, reflection can also be used to provide generic implementations of the combinators. These implementations are straightforward adaptations of the ones presented in Section 5.2. Generic implementations for `fcloseAlg` can be defined using similar techniques.

Client Code. With all self-reference infrastructure and the suitably adapted generic combinators, client code can be developed almost as before. For example:

```
val o =
    exp(fclose(merge(LiftEP,decorate(ExpEval,TraceEval.wrap),ExpPrint3)))
println("Eval: " + o.eval() + "\nPrint: " + o.print())
```

composes the variant of printing using object and family self-references and decorates evaluation with tracing. Note that this code assumes adapted versions of `LiftEP` and `ExpEval`, which are straightforward to define. The main difference to previous code is that the `fclose` operation must be applied to the composition to bind the self references and be used by the builder method `exp`. Because of the use of family and object self references tracing is applied at each call of evaluation, which ensures the expected behavior for the example.

7 Case Studies

To exercise the expressivity of feature-oriented programming using object-algebra combinators we have performed two case studies. The first adapts an example by Prehofer [43], consisting of a stack feature, which optionally can be composed with counting, locking and bounds-checking features. The second involves various interpretations and analyses of context-free grammars.

Stacks. In the first case study, we consider four features: Stack, Counter, Lock and Bound. The Stack feature captures basic stack functionality, such as `push`, `pop`, etc. If the size of stack should be maintained, the Counter feature can be used. The Lock feature prevents modifications to an entity. Finally, the Bound feature checks that some numeric input value is within bounds.

Each feature is implemented as an object algebra for stacks containing a single constructor `stack()`. If we consider the Stack feature to be the base feature, there are $2^3 = 8$ possible configurations. Each configuration requires a lifter to resolve feature interaction. For instance, lifting the counter feature to stack context involves modifying `push` and `pop` to increment and decrement the counter.

Many of these lifters require boilerplate for the methods without feature interaction. To avoid duplicating this code we have introduced default "delegating" traits. These traits declare a field for the delegatee object and forward each feature method to that object.

An example of such a delegator trait for the Counter feature is the following:

```
trait DCounter extends Counter {
  val ct: Counter
  def reset() { ct.reset }
  def inc() { ct.inc }
  def dec() { ct.dec }
  def size() = ct.size()
}
```

This trait is included, for instance, in the class that lifts the Counter feature to the Stack context, as shown in Fig. 15. The default behavior is to delegate the feature methods (e.g., `inc`, `push2`, etc.) to `Stack s` and `Counter c` respectively. To

```
class StackWithCounter(self: => Stack with Counter, s: Stack, c: Counter)
  extends DStack with DCounter {
  val st = s; val ct = c
  override def empty() {self.reset; st.empty}
  override def push(s : Char) {self.inc; st.push(s)}
  override def pop() {self.dec; st.pop}
}

object LiftSCF extends Lifter[Stack,Counter, Stack with Counter] {
  def lift(st : Stack, ct : Counter) =
    self => new StackWithCounter(self, st, ct)
}
```

Fig. 15. Lifting the Counter feature to the Stack context

```
trait GrammarAlg[In, Out] {
  def Alt(lhs: In, rhs: In): Out
  def Seq(lhs: In, rhs: In): Out
  def Opt(arg: In): Out
  def Terminal(word: String): Out
  def NonTerminal(name: String): Out
  def Empty(): Out
}
```

Fig. 16. The interface for grammar algebras

resolve feature interactions, however, the class `StackWithCounter` overrides the relevant methods, to customize the default behavior.

For feature methods that would normally call another method using **`this`**, the explicit `self` reference should be used instead. An example where this is the case is the method `push2` in the Stack feature, which calls `push` twice:

```
def push2(a : Char) {self.push(a); self.push(a)}
```

If `push2` called `push` on **`this`**, extensions of `push` would be missed, resulting, for example, in erroneous counting behavior when Stack is composed with Counter.

Grammars. The second case study implements various interpretations of grammars. The interface of grammar algebras (shown in Fig. 16) contains constructors for alternative, sequential and optional composition, terminals, non-terminals and empty. We have implemented parsing, printing, and computing nullability and first-set of a grammar symbol as individual object algebras. Parsing requires special memoization which is applied using the `decorate` combinator. In a similar way, both nullability and first-set computation require decoration with an iterative fixpoint aspect. Furthermore, the first-set feature is always composed with the nullability feature, since the former is dependent on the latter.

Tracing and profiling parsers are obtained by dynamically composing with those aspects if so desired. The tracing feature is homogeneous in that it applies uniformly to all constructors of an algebra. Profiling depends on the parsing

feature and therefore requires an explicit lifter to modify the parsing feature to update a counter. There is no feature interaction among the other features, so the default lifters produced by `combine` are sufficient.

The grammar case study illustrates the use of both object and family self-references. First, the optional constructor (`Opt`) in the interface of grammars can be desugared to an alternative composition with empty: `Opt(x)` = `fself.Alt(x, fself.Empty)`. Any grammar algebra that includes this desugaring does not have to explicitly deal with optional grammar symbols. The desugaring uses the family self-reference `fself` because the resulting term should be in the outermost, composed algebra, not the local one.

Object self-references play an important role in the profiling feature. The profiling feature accumulates a map recording the number of parse invocations on a certain grammar symbol. Using **`this`** to key into this map would create map entries on objects created by the inner, local object algebra. As a result, the objects stored in the map are without the features that may have been wrapped around these objects. For instance, in the composition `combine(combine(parse, profile), print)`, the keys in the profile map would not be printable.

Client Code. The following code creates a composite object algebra for grammars that includes parsing, nullability and first-set computation:

```
val f = fclose(
 combine[Parse, Nullable with First, Parse with Nullable with First](
  decorate(grammarParse, new Memo),
  combine[Nullable, First, Parse with Nullable with First](
   decorate(grammarNullable, new CircNullable),
   decorate(grammarFirst, new CircFirst))
  ))
```

The example shows how the three base algebras (`grammarNullable`, `grammarFirst`, and `grammarParse`) are first decorated with fixpointing and memoization behaviors. The resulting algebras are then composed using two invocations of `combine`.

The algebra `f` creates grammars with all three features built in:

```
// A ::= | "a" A
val g = Map("A" ->
   f.Alt(f.Empty, f.Seq(f.Terminal("a"), f.NonTerminal("A"))))
val s = g("A") // start symbol
s.parse(g, Seq("a", "a"), x => println("Yes"))
s.first(g) // -> Set("a")
s.nullable(g) // -> true
```

To look up non-terminals, all methods on grammars (`parse`, `first`, and `nullable`) receive the complete grammar as their first argument. The `parse` method furthermore gets an input sequence and a continuation that is called upon success.

8 Related Work

Generally speaking what distinguishes our work from previous work is the support for an expressive form of dynamic FOP that: 1) fully supports modular

type-checking and separate compilation; 2) uses only well-studied, lightweight language features which are readily available in existing languages (Scala); 3) has very small composition overhead. In contrast most existing work either uses static forms of composition, which requires heavier language features and often has significant composition overhead; requires completely new languages/calculi; or does not support modular type-checking and separate compilation.

Feature-Oriented Programming. To overcome the limitations of existing programming languages many of the existing FOP techniques [2–4, 6, 28, 43, 47] use their own language mechanisms and tools. *Mixin layers* [47] and extensions like *aspectual mixin layers* [4] are examples of language mechanisms typically used by such tools. Conceptually, mixin layers and aspectual mixin layers are quite close to our composition mechanisms, since our delegation-based model of composition has much of the same layering flavor. A difference is that mixin layers use static composition whereas we use run-time composition. As discussed throughout the paper, the `Lifter` interface used by the `merge` combinator plays a similar role to lifters in Prehofer's FOP approach [43]. Most of these language mechanisms and tools are implemented through code-generation techniques, which generally leads to an efficient and easy implementation strategy. However, this easy implementation strategy often comes at the cost of desirable properties such as separate compilation and modular type-checking. In contrast our object algebra based approach is fully integrated in an existing general purpose language (Scala); uses only already available and well-studied language features; and has full support for separate compilation and modular type-checking. The main drawback of our run-time composition mechanisms is probably performance, since delegation adds overhead which is difficult to eliminate.

More recently, researchers have also developed calculi for languages that support FOP and variants of it [1, 16, 46, 49]. These languages and calculi deal with import concerns such as type-checking or program analysis of all possible feature configurations. New languages developed for FOP typically provide novel language constructs that make features and their composition a part of the language. In contrast, our approach is to reuse existing programming language technology and to model features and feature composition with existing OO concepts. An advantage of our approach is that by using standard programming language technology all the infrastructure (type-checking, program analysis, testing, tool support) that has been developed and studied throughout the years for that language is immediately available.

Family Polymorphism. Our work can be seen as an approach to *family polymorphism* [17], but it has significantly different characteristics from most existing approaches. Traditional notions of family polymorphism are based on the idea of families grouping complete *class* hierarchies and using *static*, inheritance-based composition mechanisms. Most mechanisms used for family polymorphism, such as *virtual classes* [19], or *virtual types* [9] follow that traditional approach. In contrast our work interprets family polymorphism at the level of *objects* instead of *classes* and uses *run-time*, delegation-based composition mechanisms. The

language **gbeta** supports a type-safe *dynamic* multiple inheritance mechanism that can be used to compose families of classes [18]. Ostermann's *delegation layers* [42] model families of objects which are composed with run-time delegation. Both mechanisms are conceptually closer to our approach, but they require substantial programming language and run-time support.

Not many mechanisms that support family polymorphism are available in existing production languages. The CAKE pattern [34, 54], in the Scala programming language, is an exception. This pattern uses *virtual types*, *self types*, *path-dependent types* and (static) *mixin composition* to model family polymorphism. Even with so many sophisticated features, composition of families is quite heavyweight and manual. A particularly pressing problem, which has been acknowledged by the authors of Scala [54], is the lack of a desirable feature for family polymorphism: *deep* mixin composition. The lack of this feature means that all classes within a family have to be manually composed and then, the family itself has to be composed. In contrast, combinators like `merge` or `combine` take care of the composition of objects and the family.

Object Algebras, Visitors, Embedded DSLs and Church Encodings. Section 2 already discusses how this work addresses the problem of limited expressiveness and large composition overhead required on previous work on object algebras. Object algebras are an evolution of a line of work which exploits Church encodings of datatypes [7] to overcome modularity and extensibility issues [24,38,39]. These techniques were later shown to be related to the VISITOR pattern and used to provide modular and generic visitors [36, 41]. They have also been successfully used to represent embedded DSLs in more modular ways [10,25]. However, most existing work considers only the creation of objects with single features: not much attention is paid to creating objects composed of multiple features. Hofer et al. [25] use delegation in an optimization example. Their use of delegation is analogous to our use of dependent features in the printing operation presented in Section 6.2. However we express dependencies using a bound on the self-reference type, and use `merge` in client code to compose features. Their approach is more manual as they have to delegate behavior for each case. The generalization of object algebra interfaces in Fig. 10 was first used by Oliveira et al. [41] to provide a unified interface for visitor interfaces. Oliveira [36] also explored a kind of FOP using modular visitors. However there is a lot of overhead involved to create feature modules and compose features; and the approach requires advanced language features and does not deal with self-references.

9 Conclusion

Feature-oriented programming is an attractive programming paradigm. However it has been traditionally difficult to provide language mechanisms with the benefits of FOP, while at the same time having desirable properties like separate compilation and modular type-checking.

This work shows that it is possible to support FOP with such desirable properties using existing language mechanisms and OO abstractions. To accomplish this

we build on previous work on object algebras and two well-studied studied programming language features: *intersection types* and *type-constructor polymorphism*. Object algebras provide the basic support for modularity and extensibility to coexist with separate compilation and modular type-checking. Intersection types and type-constructor polymorphism provide support for the development of safe and expressive composition mechanisms for object algebras. With those composition mechanisms, expressing *feature interactions* becomes possible, thus enabling support for FOP.

Although we have promoted the use of standard programming language technology for FOP, there is still a lot to be gained from investigating new programming language technology to improve our results. Clearly the investigation of better compilation techniques for object algebras and composition mechanisms is desirable for improving performance. Better language support for delegation would allow for more convenient mechanisms for object-level reuse, which are often needed with our techniques. Expressiveness could also be improved with new programming languages or extensions. For example when multi-sorted object algebras [37] are required the `Algebra` interfaces still have to be adapted. Although more powerful generic programming techniques can address this problem, this requires more sophisticated general purpose language features. With built-in support for object algebras as well as their composition mechanisms, there is no need for such advanced generic programming features and users may benefit from improved support for error messages.

Acknowledgements. We are grateful to the anonymous reviewers for various comments and suggestions which have helped improving this paper significantly.

References

1. Apel, S., Kästner, C., Grösslinger, A., Lengauer, C.: Type safety for feature-oriented product lines. Automated Software Engg. 17(3) (September 2010)
2. Apel, S., Kastner, C., Lengauer, C.: Featurehouse: Language-independent, automated software composition. In: ICSE 2009 (2009)
3. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 125–140. Springer, Heidelberg (2005)
4. Apel, S., Leich, T., Saake, G.: Aspectual mixin layers: aspects and features in concert. In: ICSE 2006 (2006)
5. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology 8(5), 49–84 (2009)
6. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. on Softw. Eng. 30(6), 355–371 (2004)
7. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theor. Comput. Sci., 135–154 (1985)
8. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: OOPSLA 2004 (2004)
9. 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)

10. 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)
11. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for java. In: OOPSLA 2000 (2000)
12. Compagnoni, A.B., Pierce, B.C.: Higher-order intersection types and multiple inheritance. Math. Structures Comput. Sci. 6(5), 469–501 (1996)
13. Cook, W.R., Palsberg, J.: A denotational semantics of inheritance and its correctness. In: OOPSLA 1989 (1989)
14. 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)
15. Coppo, M., Dezani-Ciancaglini, M.: A new type-assignment for λ-terms. Archiv. Math. Logik 19, 139–156 (1978)
16. Damiani, F., Padovani, L., Schaefer, I.: A formal foundation for dynamic delta-oriented software product lines. In: GPCE 2012 (2012)
17. Ernst, E.: Family polymorphism. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
18. Ernst, E.: Safe dynamic multiple inheritance. Nordic J. of Computing 9(3) (2002)
19. Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: POPL 2006 (2006)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addisson-Wesley professional computing series. Addisson-Wesley (1994)
21. van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). IEEE Computer Society (2001)
22. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica (1978)
23. Harrison, W., Ossher, H.: Subject-oriented programming (A critique of pure objects). In: OOPSLA 1993 (1993)
24. Hinze, R.: Generics for the masses. Journal of Functional Programming 16(4-5), 451–483 (2006)
25. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of DSLs. In: GPCE 2008 (2008)
26. Jones, P., Wollrath, A., Scheifler, R., et al.: Method and system for dynamic proxy classes, US Patent 6,877,163 (2005)
27. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
28. Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating support for features in advanced modularization technologies. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 169–194. Springer, Heidelberg (2005)
29. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1), 1–13 (2008)
30. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: new-age components for old-fashioned java. In: OOPSLA 2001 (2001)
31. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: OOPSLA 2008 (2008)
32. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA 2006 (2006)

33. Odersky, M.: The Scala Language Specification, Version 2.9. EPFL (2011), http://www.scala-lang.org/docu/files/ScalaReference.pdf
34. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA 2005 (2005)
35. Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: Mri: Modular reasoning about interference in incremental programming. Journal of Functional Programming 22, 797–852 (2012)
36. Oliveira, B.C.d.S.: Modular visitor components. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 269–293. Springer, Heidelberg (2009)
37. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses: Practical extensibility with object algebras. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 2–27. Springer, Heidelberg (2012)
38. Oliveira, B.C.d.S., Gibbons, J.: Typecase: a design pattern for type-indexed functions. In: Haskell 2005 (2005)
39. Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Trends in Functional Programming (2006)
40. Oliveira, B.C.d.S., Moors, A., Odersky, M.: Type classes as objects and implicits. In: OOPSLA 2010 (2010)
41. Oliveira, B.C.d.S., Wang, M., Gibbons, J.: The visitor pattern as a reusable, generic, type-safe component. In: OOPSLA 2008 (2008)
42. Ostermann, K.: Dynamically composable collaborations with delegation layers. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 89–110. Springer, Heidelberg (2002)
43. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–443. Springer, Heidelberg (1997)
44. 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)
45. Reynolds, J.C.: Towards a theory of type structure. In: Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 408–425. Springer, Heidelberg (1974)
46. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: AOSD 2011 (2011)
47. Smaragdakis, Y., Batory, D.: Implementing layered designs with mixin layers. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 550–570. Springer, Heidelberg (1998)
48. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE 1999 (1999)
49. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: GPCE 2007 (2007)
50. Torgersen, M.: The expression problem revisited – four new solutions using generics. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
51. Wadler, P.: The Expression Problem. Email, discussion on the Java Genericity mailing list (November 1998)
52. Wehr, S., Thiemann, P.: JavaGI: The interaction of type classes with interfaces and inheritance. ACM Trans. Program. Lang. Syst. 33 (July 2011)
53. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: ICFP 2001 (2001)
54. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL 2005 (2005)

Composition and Reuse with Compiled Domain-Specific Languages

Arvind K. Sujeeth[1], Tiark Rompf[2,3], Kevin J. Brown[1], HyoukJoong Lee[1], Hassan Chafi[1,3], Victoria Popic[1], Michael Wu[1], Aleksandar Prokopec[2], Vojin Jovanovic[2], Martin Odersky[2], and Kunle Olukotun[1]

[1] Stanford University
{asujeeth,kjbrown,hyouklee,hchafi,viq,mikemwu,kunle}@stanford.edu
[2] École Polytechnique Fédérale de Lausanne (EPFL)
{firstname.lastname}@epfl.ch
[3] Oracle Labs
{firstname.lastname}@oracle.com

Abstract. Programmers who need high performance currently rely on low-level, architecture-specific programming models (e.g. OpenMP for CMPs, CUDA for GPUs, MPI for clusters). Performance optimization with these frameworks usually requires expertise in the specific programming model and a deep understanding of the target architecture. Domain-specific languages (DSLs) are a promising alternative, allowing compilers to map problem-specific abstractions directly to low-level architecture-specific programming models. However, developing DSLs is difficult, and using multiple DSLs together in a single application is even harder because existing compiled solutions do not compose together. In this paper, we present four new performance-oriented DSLs developed with Delite, an extensible DSL compilation framework. We demonstrate new techniques to compose compiled DSLs embedded in a common backend together in a single program and show that generic optimizations can be applied across the different DSL sections. Our new DSLs are implemented with a small number of reusable components (less than 9 parallel operators total) and still achieve performance up to 125x better than library implementations and at worst within 30% of optimized stand-alone DSLs. The DSLs retain good performance when composed together, and applying cross-DSL optimizations results in up to an additional 1.82x improvement.

1 Introduction

High-level general purpose languages focus on primitives for abstraction and composition that allow programmers to build large systems from relatively simple but versatile parts. However, these primitives do not usually expose the structure required for high performance on today's hardware, which is parallel and heterogeneous. Instead, programmers are forced to optimize performance-critical sections of their code using low-level, architecture-specific programming models (e.g. OpenMP, CUDA, MPI) in a time-consuming process. The optimized

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 52–78, 2013.

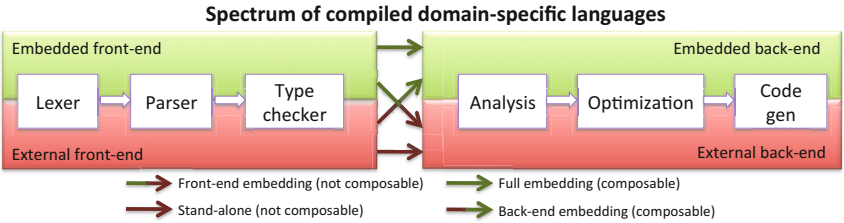


Fig. 1. Major phases in a typical compiler pipeline and possible organizations of compiled DSLs. Front-end embedding in a host language (and compiler) is common, but for composability, back-end embedding in a host compiler (i.e. building on top of an extensible compiler framework) is more important.

low-level code is harder to read and maintain, more likely to contain hard-to-diagnose bugs, and difficult to port to other platforms or hardware.

Domain-specific languages (DSLs) have been proposed as a solution that can provide productivity, performance, and portability for high-level programs in a specific domain [1]. DSL compilers reason about a program at the level of domain operations and so have much more semantic knowledge than a general purpose compiler. This semantic knowledge enables coarse-grain optimizations and the translation of domain operations to efficient back-end implementations. However, the limited scope of DSLs is simultaneously a stumbling block for widespread adoption. Many applications contain a mix of problems in different domains and developers need to be able to compose solutions together without sacrificing performance. In addition, DSLs need to interoperate with the outside world in order to enable developers to fall-back to general purpose languages for non performance-critical tasks.

Existing implementation choices for DSLs range from the internal (i.e. purely embedded) to external (i.e. stand-alone). Purely embedded DSLs are implemented as libraries in a flexible host language and emulate domain-specific syntax. The main benefit of internal DSLs is that they are easy to build and compose, since they can interoperate freely within the host language. However, they do not achieve high performance since the library implementation is essentially an interpreted DSL with high overhead and since general purpose host languages do not target heterogeneous hardware. On the other end of the spectrum, stand-alone DSLs are implemented with an entirely new compiler that performs both front-end tasks such as parsing and type checking as well as back-end tasks like optimization and code generation. Recent work has demonstrated that stand-alone DSLs can target multiple accelerators from a single source code and still achieve performance comparable to hand-optimized versions [2,3]. The trade-off is that each DSL requires a huge amount of effort that is not easy to reuse in other domains, DSL authors must continuously invest new effort to target new hardware, and DSL programs do not easily interoperate with non-DSL code.

Another class of DSLs, compiled embedded, occupies a middle-ground between the internal and external approaches [4,5,6,7]. These DSLs embed their front-end in a host language like internal DSLs, but use compile- or run-time code generation

to optimize the embedded code. Recently, these techniques have also been used to generate not only optimized host language code, but heterogeneous code, e.g. targeted at GPGPUs. The advantage of this approach is that the languages are easier to build than external versions and can provide better performance than internal versions. However, these DSLs still give up the composability of purely embedded DSLs and the syntactic freedom of stand-alone DSLs.

Like previous compiled embedded DSLs, our goal is to construct DSLs that resemble self-optimizing libraries with domain-specific front-ends. However, we propose that to build high-performance composable DSLs, *back-end embedding* is more important than the *front-end embedding* of traditional compiled embedded DSLs. Figure 1 illustrates this distinction. We define back-end embedding as a compiled DSL that inherits, or extends, the latter portion of the compiler pipeline from an existing compiler. Such a DSL can either programmatically extend the back-end compiler or pass programs in its intermediate representation (IR) to the back-end compiler. By embedding multiple DSLs in a common back-end, they can be compiled together and co-optimized. The fact that many compiled DSLs target C or LLVM instead of machine code and thus reuse instruction scheduling and register allocation can be seen as a crude example of back-end embedding. However, the abstraction level of C is already too low to compose DSLs in a way that allows high-level cross-DSL optimizations, parallelization, or heterogeneous code generation. Just as some general purpose languages are better suited for front-end embedding than others, not all compiler frameworks or target languages are equally suited for back-end embedding.

In this paper, we show that we can compose compiled DSLs embedded in a common, high-level backend and use them together in a single application. Our approach allows DSLs to build on top of one another and reuse important generic optimizations, rather than reinventing the wheel each time. Optimizations can even be applied across different DSL blocks within an application. The addition of composablity and re-use across compiled DSLs pushes them closer to libraries in terms of development effort and usage while still retaining the performance characteristics of stand-alone DSLs. In other words, we regain many of the benefits of purely embedded DSLs that were lost when adding compilation. We build on our previous work, Lightweight Modular Staging (LMS) and Delite [8,9,10], frameworks designed to make constructing individual compiled embedded DSLs easier. Previous work demonstrated good performance for OptiML, a DSL for machine learning [11]. However, it did not address how to compose different DSLs together, or show that similar performance and productivity gains could be obtained for different domains. We present new compiled embedded DSLs for data querying (OptiQL), collections (OptiCollections), graph analysis (OptiGraph), and mesh computation (OptiMesh). We show that the DSLs were easier to build than stand-alone counterparts, can achieve competitive performance, and can also be composed together in multiple ways.

Specifically, we make the following contributions:

– We implement four new DSLs for different domains and show that they can be implemented with a small number of reusable components and still achieve

performance exceeding optimized libraries (up to 125x) and comparable to stand-alone DSLs (within 30%).
- We are the first to show both fine-grained and coarse-grained composition of high performance compiled DSLs.
- We demonstrate that different DSLs used in the same application can be co-optimized to additionally improve performance by up to 1.82x.

The source code for the new DSLs we have developed is open-source and freely available at: `http://github.com/stanford-ppl/Delite/`.

2 Background

In this paper, we investigate the problem of composing different DSLs embedded in a common back-end. The back-end we will use for our examples is Delite, but any back-end with similar structure could use the same techniques. Similarly, while Delite is typically targeted from embedded Scala front-ends, it could also be targeted from an external parser. Delite is essentially a Scala library that DSL authors can use to build an intermediate representation (IR), perform optimizations, and generate parallel code for multiple hardware targets. To illustrate at a high-level how Delite works, consider a simple example of a program using a Delite Vector DSL to add two vectors:

```
val (v1,v2) = (Vector.rand(1000), Vector.rand(1000))
val a = v1+v2
println(a)
```

The DSL implementation maps each language statement (`Vector.rand`, `+`, `println`) in this program to parallel operators (ops), each of which represents a specific parallel pattern (e.g. map, reduce, fork/join, sequential). These patterns are provided by Delite and extended by the DSL. The mapping is accomplished using a technique called Lightweight Modular Staging (LMS), a form of *staged metaprogramming* [12]. The essence is that the DSL implements operations on types wrapped in an abstract type constructor, `Rep[T]`. Type inference is used to hide this wrapped type from application code, as in the above snippet. Instead of immediate evaluation, DSL operations on `Rep[T]` construct an IR node representing the operation. For example, when the statement `v1+v2` is executed, it actually calls a DSL method that constructs a Delite IR node and returns a well-typed placeholder (`Rep[Vector[Double]]`) for the result. To ensure that all host language operations can be intercepted and lifted, we use a modified version of the Scala compiler for front-end compilation, *Scala-Virtualized* [13], that enables overloading even built-in Scala constructs such as `if (c) a else b`. DSLs can perform domain-specific optimizations by traversing and transforming the IR; Delite uses the same mechanisms to perform generic optimizations (such as dead code elimination) for all DSLs. Finally, after the full IR has been constructed and optimized, Delite generates code for each operation, based on its parallel pattern, to multiple targets (Scala, C++, CUDA). The resulting generated code is executed in a separate step to compute the final answer.

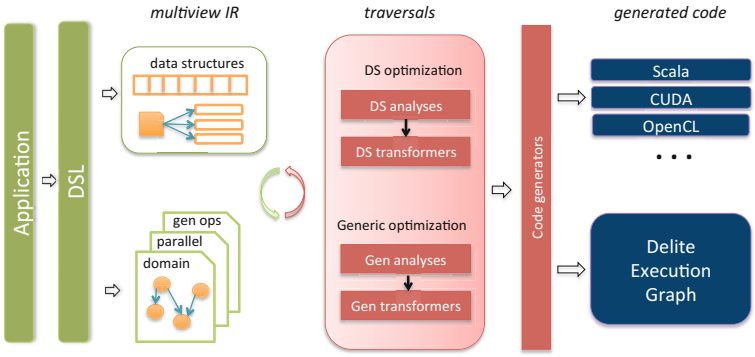


Fig. 2. Components of the Delite Framework. An application is written in a DSL, which is composed of data structures and structured computations represented as a multi-view IR. The IR is transformed by iterating over a set of traversals for both generic (Gen) and domain-specific (DS) optimizations. Once the IR is optimized, heterogeneous code generators emit specialized data structures and ops for each target along with the Delite Execution Graph (DEG) that encodes dependencies between computations.

Figure 2 illustrates the key reusable components of the Delite compiler architecture: common IR nodes, data structures, parallel operators, built-in optimizations, traversals, transformers, and code generators. DSLs are developed by extending these reusable components with domain-specific semantics. Furthermore, Delite is modular; any service it provides can be overridden by a particular DSL with a more customized implementation.

3 High-Level Common Intermediate Representation

To achieve high performance for an application composed out of multiple DSLs, each DSL must provide competitive performance, not incur high overhead when crossing between DSL sections, and be co-optimizable. Moreover, for this sort of composition to be a practical approach, DSLs targeting narrow problem domains and modern hardware should be as easy to construct as optimized libraries (or as close as possible). One way of achieving these goals is through a common intermediate representation containing reusable, high level computation and data primitives with a minimal set of restrictions that enable efficiency and optimization. In this section we describe how the Delite IR meets this criteria and propose a simple data exchange format for embedded DSLs.

3.1 Structured Computation

In order to optimize composed DSL blocks, they must share, or be transformable to, a common intermediate representation (IR) at some level. This level should retain enough high-level semantics to allow coarse-grained optimization and code generation; once an operation has been lowered to low-level primitives such as

instructions over scalars, it is as difficult to target heterogeneous hardware as in a general purpose language. If DSLs do not share a common IR at some level, the best we can do is attempt to compose their generated code, which is a low-level and error-prone strategy that does not allow co-optimization.

We have proposed parallel patterns as a base IR that is well-suited to optimization and code generation for different DSLs and heterogeneous devices [8]. The new DSLs presented in this paper offer more evidence that parallel patterns are a strong choice of base IR. Previously, Delite supported `Sequential`, `Loop`, `Map`, `Reduce`, `ZipWith`, `Foreach`, and `Filter` patterns. To support the new DSLs, we added support for `GroupBy`, `Join` (generic inner join), `Sort`, `ForeachReduce` (foreach with global reductions), and `FlatMap` patterns. Using the patterns as a base IR allows us to perform *op fusion* automatically for each DSL, which is a key optimization when targeting data parallel hardware such as GPUs.

3.2 Structured Data

In the same way that a common, structured IR allows DSL operations to be optimized and code generated in a uniform way, a common data representation is also required at some level. In addition to enabling the internal representation of DSL data structures to be reusable, a common representation facilitates communication between DSL blocks and enables pipeline optimizations; a DSL can directly consume the output of another DSL's block, so we can (for example) fuse the first DSL's operation that constructs the output with the second DSL's operation that consumes it.

We use structs of a fixed set of primitives as the common data representation. Delite DSLs still present domain-specific types to end users (e.g. `Vector`) and methods on instances of those types (e.g. `+`) get lifted into the IR. However, the back-end data structures must be implemented as structs (for example, `Vector` can be implemented with an `Int` length field and an `Array` data field). The fields currently allowed in a Delite `Struct` are: numerics (e.g. `Int`), `Boolean`, `String`, `Array`, `HashMap`, or other `Structs`. By restricting the content of data structures, Delite is able to perform optimizations such as array-of-struct (AoS) to struct-of-array (SoA) conversion and dead field elimination (DFE) automatically [14]. Furthermore, since the set of primitives is fixed, Delite can implement these primitives on each target platform (e.g. C++, CUDA) and automatically generate code for DSL structs. Delite also supports user-defined data structures by lifting the **`new`** keyword defining an anonymous class [13]. An application developer can write code like the following:

```
val foo = new Record { val x = "bar"; val y = 42 }
```

`Record` is a built-in type provided by Delite that serves as a tag for the Scala-Virtualized compiler. The Scala-Virtualized compiler forwards any invocation of **`new`** with a `Record` type to Delite, which will then construct a corresponding `Struct`. Field accesses to the record are type-checked by the Scala-Virtualized compiler. In this way, all DSL and user types can uniformly be lowered to one of the primitives, or a `Struct` of primitives.

3.3 Data Exchange Format

The final piece required for composability is the ability for application developers to convert from domain-specific data types to common data types in order to communicate between DSL blocks. This is necessary because we do not want to expose the internal representation of the domain-specific types to users.

A simple solution that we use is for DSL authors to optionally implement methods `to/from{Primitive}` between each DSL data type and a corresponding primitive type. For example, a DSL author would provide `toArray` and `fromArray` methods on DSL types where this makes sense (e.g. `Vector`), or `toInt` and `fromInt` for a type `NodeId`. These methods become part of the DSL specification and enable application developers to export and import DSL objects depending on their needs. For example, a graph could be exported using a snippet like:

```
// G: Graph
// returns an array of node ids and an array of edge ids
new Record { val nodes = G.nodes.map(node => node.Id.toInt).toArray
             val edges = G.edges.map(edge => edge.Id.toInt).toArray }
```

or just one of the properties of the graph could be exported using:

```
// G: Graph, numFriends: NodeProperty[Int]
// returns an array of ints corresponding to the number of friends of each node
G.nodes.map(node => numFriends(node)).toArray
```

We consider in the next section how to actually compose snippets together. Assuming this facility, though, importing is similar. For example, a vector could be constructed from the output of the previous snippet using:

```
// numFriends: Array[Int]
val v = Vector.fromArray(numFriends)
```

The key aspect of the data exchange format is that it should not prevent optimization across DSL blocks or impose substantial overhead to box and unbox the results. We handle this by implementing the `to/from` functions as either scalar conversions or loops in the common IR. Then, for example, a loop constructing an array will be automatically fused together with the loop consuming the array, resulting in no overhead while still ensuring safe encapsulation. The loop fusion algorithm is described in previous work [14].

4 Methods for Composing Compiled DSLs

In this section, we describe two ways of composing compiled DSLs that are embedded in a common back-end. The first way is to combine DSLs that are designed to work with other DSLs, i.e. DSLs that make an "open-world" assumption. The second way is to compose "closed-world" DSLs by compiling them in isolation, lowering them to a common, high-level representation, recombining them and then optimizing across them. Both methods rely on DSLs that share a common high-level intermediate representation as described in the previous section.

4.1 Open-World: Fine-Grained Cooperative Composition

"Open-world" composition means essentially designing embedded DSLs that are meant to be layered or included by other embedded DSLs as modules. For this kind of composition to be feasible, all of the DSLs must be embedded in the same language and framework in both the front-end and the back-end.

Once embedded in a common host environment, DSL composition reduces to object composition in the host language. This is the classic "modularity" aspect of DSLs built using LMS [9]. For example, consider the following DSL definition for a toy DSL `Vectors` embedded in Scala:

```
trait Vectors extends Base with MathOps with ScalarOps with VectorOps
```

The DSL is composed of several traits that contain different DSL data types and operations. Each trait extends `Base` which contains common IR node definitions. A different DSL author can extend `Vectors` to create a new DSL, `Matrices`, simply by extending the relevant packages:

```
trait Matrices extends Vectors with MatrixOps with ExtVectorOps
```

Each of the mixed-in traits represents a collection of IR node definitions. Traits that contain optimizations and code generators can be extended in the same way. These traits define `Matrices` as a superset of `Vectors`, but `Matrices` users only interact with `Matrices` and do not need to be aware of the existence of `Vectors`. Furthermore, since the composition is via inheritance, the `Matrices` DSL can extend or overload operations on the `Vector` data type, e.g. inside `ExtVectorOps`. Since `Vectors` is encapsulated as a separate object, it can be reused by multiple DSLs.

Open-world DSLs can also be composed by application developers. For example, suppose we also have a visualization DSL:

```
trait Viz extends Base with GraphicsOps with ChartOps with ImageOps
```

A Scala program can effectively construct a new DSL on the fly by mixing multiple embedded DSLs together:

```
trait MyApp extends Matrices with Viz {
  def main() {
    // assuming relevant DSL functions
    val m = Matrix.rand(100); display(m.toArray)
  }
}
```

In this example we make use of the data exchange format described in Section 3.3 in order to communicate data between the DSLs. When DSL users invoke a `Viz` operation, that operation will construct a `Viz` IR node. Note that after mix-in, the result is effectively a single DSL that extends common IR nodes; optimizations that operate on the generic IR can occur even between operations from different DSLs. This is analogous to libraries building upon other libraries, except that now optimizing compilation can also be inherited. DSLs can add analyses and transformations that are designed to be included by other DSLs. The trade-off is that DSL authors and application developers must be aware of the semantics they are composing and are responsible for ensuring that

transformations and optimizations between DSLs retain their original semantics. Namespacing can also be a pitfall; DSL traits cannot have conflicting object or method names since they are mixed in to the same object. We avoid this problem by using conventions like DSL-specific prefixes (e.g. `viz_display`).

4.2 Closed-World: Coarse-Grained Isolated Composition

As the previous section pointed out, there are issues with simply mixing embedded DSLs together. In particular, some DSLs require restricted semantics in order to perform domain-specific analyses and transformations that are required for correctness or performance. These "closed-world" DSLs are not designed to arbitrarily compose with other DSLs. Furthermore, any DSL with an external front-end is necessarily closed-world, since the parser only handles that DSL's grammar. However, it is still possible to compose closed-world DSLs that share a common back-end. Either they are implemented in the same language as the back-end and programmatically interface with it (as with Scala and Delite), or they target a serialized representation (such as source code) that is the input to a common back-end. This kind of coarse grained composition has three steps:

1. Independently parse each DSL block and apply domain-specific optimizations
2. Lower each block to the common high-level IR, composing all blocks into a single intermediate program
3. Optimize the combined IR and generate code

We discuss how we implemented these steps in Delite next.

Scopes: independent compilation DSLs that have an external front-end can be independently compiled by invoking the DSL parser on a string. However, in order to independently compile DSL blocks that are embedded in a host language, we need a coarse-grained execution block. We have modified the Scala-Virtualized compiler to add `Scopes` for this purpose. A `Scope` is a compiler-provided type that acts as a tag to encapsulate DSL blocks. For example, using the `Vectors` DSL from the previous section, we can instantiate a `Scope` as follows:

```
def Vectors[R](b: => R) = new Scope[VectorsApp, VectorsCompiler, R](b)
```

`VectorsApp` and `VectorsCompiler` are Scala traits that define the DSL interface and its implementation, respectively. The Scala-Virtualized compiler transforms function calls with return type `Scope` into an object that composes the two traits with the given block, making all members of the DSL interface available to the block's content. The implementation of the DSL interface remains hidden, however, to ensure safe encapsulation. The object's constructor then executes the block. The result is that each `Scope` is staged and optimized independently to construct the domain-specific IR.

Given the previous definition, a programmer can write a block of `Vectors` DSL code inside a Scala application, which then gets desugared, making all member definitions of `VectorsApp`, but not of `VectorsCompiler`, available to the `Scope`'s body:

```
Vectors {
  val v = Vector.rand(100)
  // ...
}
```

(a) Scala code with DSL scope

```
abstract class DSLprog extends VectorsApp {
  def apply = {
    val v = Vector.rand(100)
    // ...
  }
}
(new DSLprog with VectorsCompiler).result
```

(b) Scala code after desugaring

Lowering and Composing. The ability to compile blocks of DSL code into independent IRs is the first step, but in order to compose multiple blocks in a single application we still need a way to communicate across the blocks and a way to combine the IRs. Consider the following application snippet:

```
val r = Vectors {
  val (v1,v2) = (Vector.rand(100),Vector.rand(100))
  DRef(linreg(v1,v2).toArray) // return linear regression of v1, v2
}
Viz { display(r.get) }
```

We again use the `toArray` functionality to export the data from `Vectors` into a common format that `Viz` can handle. However, before we lower to a common representation, the type of the output of `Vectors` is a symbol with no relation to `Viz`. Therefore, we introduce the path independent type `DRef[T]` to abstract over the path dependent scope result. During staging, `r.get` returns a placeholder of type `Rep[DeliteArray[Double]]`. When the IRs are lowered and stitched together, the placeholder is translated to the concrete symbolic result of `linreg(v1,v2).toArray`. This mechanism is type-safe, preserves scope isolation, and does not occlude optimizations on the lowered IR.

After performing domain-specific optimizations and transformations, the IR for each scope is lowered to the base IR in a language-specific fashion. We use staging to perform this translation by extending our previous work on staged interpreters for program transformation [14] to support transforming IR nodes to an arbitrary target type. A `Transformer` is a generic `Traversal` that maps symbolic IR values (type `Exp[A]`) to values of type `Target[A]`, where `Target` is an abstract type constructor. During the traversal, a callback `transformStm` is invoked for each statement encountered.

The extended `Transformer` interface for cross-DSL transformation is:

```
trait Transformer extends Traversal {
  import IR._
  type Target[A]
  var subst = immutable.Map.empty[Exp[Any], Target[Any]]
  def apply[A](x: Exp[A]): Target[A] = ...    // lookup from subst
  override def traverseStm(stm: Stm): Unit =  // called during traversal
    subst += (stm.sym -> transformStm(stm))   //   update substitution with result
  def transformStm(stm: Stm): Target[Any]     // to be implemented in subclass
}
```

To transform from one IR to another, lower-level IR language, we instantiate a transformer with type `Target` set to the `Rep` type of the destination IR:

```
trait IRTransformer extends Transformer {
  val dst: IR.DST
  type Target[A] = IR.Rep[A]
  def transformStm(stm: Stm): Target[Any] =   // override to implement custom transform
    IR.mirror(stm.rhs, this)                  //   call default case
}
```

The type of the destination IR `dst: IR.DST` is constrained by the source IR to handle all defined IR nodes. This enables implementing a default case for the transformation (**def** `mirror`), which maps each source IR node to the corresponding smart constructor in the destination IR.

Taking the Vectors DSL as an example, we define:

```
trait Vectors extends Base {
  def vector_zeros(n: Rep[Int]): Rep[Vector[Double]]
}
trait VectorExp extends Vectors with BaseExp {
  type DST <: Vectors
  case class VectorZeros(n: Rep[Int]) extends Def[Vector[Double]]
  def vector_zeros(n: Rep[Int]): Rep[Vector[Double]] = VectorZeros(n)
  def mirror[A:Manifest](e: Def[A], f: IRTransformer): f.dst.Rep[A] = {
    case VectorZeros(n) => f.dst.vector_zeros(f(n))
    ...
  }
}
```

The use of Scala's dependent method types `f.dst.Rep[A]` and the upper-bounded abstract type `DST <: Vectors` ensure type safety when specifying transformations. Note that the internal representation of the destination IR is not exposed, only its abstract `Rep` interface. This enables, for example, interfacing with a textual code generator that defines `Rep[T] = String`. Perhaps more importantly, this enables programmatic lowering transforms by implementing a smart constructor (e.g. `vector_zeros`) to expand into a lower-level representation using arrays instead of constructing an IR node that directly represents the high-level operation.

An alternative to using staged interpreters is to simply generate code to a high-level intermediate language that maps to the common IR. We implemented this by generating code to the "Delite IL", a low-level API around Delite ops, that is itself staged to build the Delite IR. For example, a `Reduce` op in the original application would be code generated to call the method

```
reduce[A](size: Rep[Int], func: Rep[Int] => Rep[A], cond: List[Rep[Int]] => Rep[Boolean],
          zero: => Rep[A], rFunc: (Rep[A],Rep[A]) => Rep[A])
```

in the Delite IL.

The staged interpreter transformation and the code generation to the Delite IL perform the same operation and both use staging to build the common IR. The staged interpreter translation is type-safe and goes through the heap, while the Delite IL goes through the file system and is only type-checked when the

resulting program is (re-)staged. On the other hand, for expert programmers, a simplified version of the Delite IL may be desirable to target directly.

Cross DSL Optimization. After the IRs have been composed, we apply all of our generic optimizations on the base IR (i.e. parallel patterns). Like in the open-world case, we can now apply optimizations such as fusion, common subexpression elimination (CSE), dead code elimination (DCE), dead field elimination (DFE), and AoS to SoA conversion across DSL snippets. Since the base IR still represents high level computation, these generic optimizations still have much higher potential impact than their analogs in a general purpose compiler. Fusion across DSL snippets is especially useful, since it can eliminate the overhead of boxing and unboxing the inputs and outputs to DSL blocks using the `to/from{Primitive}` data exchange format. The usefulness of applying these optimizations on composed blocks instead of only on individual blocks is evaluated in Section 6. Note that the cross-DSL optimizations fall out for free after composing the DSLs; we do not have to specifically design new cross-DSL optimizations.

4.3 Interoperating with Non-DSL Code

The previous section showed how we can compose coarse-grain compiled DSL blocks within a single application. However, it is also interesting to consider how we can compose DSL blocks with arbitrary host language code. We can again use `Scope`, but with a different implementation trait, to accomplish this. Consider the following, slightly modified definition of the `Vectors` scope:

```
def Vectors[R](b: => R) = new Scope[VectorsApp, VectorsExecutor, R](b)
```

Whereas previously the `Vectors` scope simply compiled the input block, the trait `VectorsExecutor` both compiles and executes it, returning a Scala object as a result of the execution. `VectorsExecutor` can be implemented by programmatically invoking the common back-end on the lowered IR immediately. This enables us to use compiled embedded DSLs within ordinary Scala programs:

```
def main(args: Array[String]) {
  foo() // Scala code
  Vectors { val v = Vector.ones(5); v.pprint }
  // more Scala code ..
}
```

This facility is the same that is required to enable interactive usage of DSLs using the REPL of the host language, which is especially useful for debugging. For example, we can use the new `Vectors` scope to execute DSL statements inside the Scala-Virtualized REPL:

```
scala> Vectors { val v = Vector.ones(5); v.pprint }
[ 1 1 1 1 1 ]
```

5 New Compiled Embedded DSL Implementations

We implemented four new high performance DSLs embedded inside Scala and Delite. In this section, we briefly describe each DSL and show how their

```
1  // lineItems: Table[LineItem]
2  val q = lineItems Where(_.l_shipdate <= Date(''1998-12-01'')).
3  GroupBy(l => (l.l_linestatus)) Select(g => new Record {
4    val lineStatus = g.key
5    val sumQty = g.Sum(_.l_quantity)
6    val sumDiscountedPrice = g.Sum(l => l.l_extendedprice*(1.0-l.l_discount))
7    val avgPrice = g.Average(_.l_extendedprice)
8    val countOrder = g.Count
9  }) OrderBy(_.returnFlag) ThenBy(_.lineStatus)
```

Listing 1.1. OptiQL: TPC-H Query 1 benchmark

implementation was simplified by reusing common components. The four new DSLs are OptiQL, a DSL for data querying, OptiCollections, an optimized subset of the Scala collections library, OptiGraph, a DSL for graph analysis based on Green-Marl [3] and OptiMesh, a DSL for mesh computations based on Liszt [2]. Despite being embedded in both a host language and common back-end, the DSLs cover a diverse set of domains with different requirements and support non-trivial optimizations.

5.1 OptiQL

OptiQL is a DSL for data querying of in-memory collections, and is heavily inspired by LINQ [15], specifically LINQ to Objects. OptiQL is a pure language that consists of a set of implicitly parallel query operators, such as `Select`, `Average`, and `GroupBy`, that operate on OptiQL's core data structure, the `Table`, which contains a user-defined schema. Listing 1.1 shows an example snippet of OptiQL code that expresses a query similar to Q1 in the TPC-H benchmark. The query first excludes any line item with a ship date that occurs after the specified date. It then groups each line item by its status. Finally, it summarizes each group by aggregating the group's line items and constructs a final result per group.

Since OptiQL is SQL-like, it is concise and has a small learning curve for many developers. However, unoptimized performance is poor. Operations always semantically produce a new result, and since the in-memory collections are typically very large, cache locality is poor and allocations are expensive. OptiQL uses compilation to aggressively optimize queries. Operations are fused into a single loop over the dataset wherever possible, eliminating temporary allocations, and datasets are internally allocated in a column-oriented manner, allowing OptiQL to avoid allocating columns that are not used in a given query. Although not implemented yet, OptiQL's eventual goal is to use Delite's pattern rewriting and transformation facilities to implement other traditional (domain-specific), cost-based query optimizations.

5.2 OptiCollections

Where OptiQL provides a SQL-like interface, OptiCollections is an example of applying the underlying optimization and compilation techniques to the Scala

```
1  val sourcedests = pagelinks flatMap { l =>
2    val sd = l.split(":")
3    val source = Long.parseLong(sd(0))
4    val dests = sd(1).trim.split(" ")
5    dests.map(d => (Integer.parseInt(d), source))
6  }
7  val inverted = sourcedests groupBy (x => x._1)
```

Listing 1.2. OptiCollections: reverse web-links benchmark

collections library. The Scala collections library provides several key generic data types (e.g. `List`) with rich APIs that include expressive functional operators such as `flatMap` and `partition`. The library enables writing succinct and powerful programs, but can also suffer from overheads associated with high-level, functional programs (especially the creation of many intermediate objects). OptiCollections uses the exact same collections API, but uses Delite to generate optimized, low-level code. Most of the infrastructure is shared with OptiQL. The prototype version of OptiCollections supports staged versions of Scala's `Array` and `HashMap`. Listing 1.2 shows an OptiCollections application that consumes a list of web pages and their outgoing links and outputs a list of web pages and the set of incoming links for each of the pages (i.e. finds the reverse web-links). In the first step, the `flatMap` operation maps each page to pairs of an outgoing link and the page. The `groupBy` operation then groups the pairs by their outgoing link, yielding a `HashMap` of pages, each paired with the collection of web pages that link to it.

The example has the same syntax as the corresponding Scala collections version. A key benefit of developing OptiCollections is that it can be mixed in to enrich other DSLs with a range of collection types and operations on those types. It can also be used as a transparent, drop-in replacement for existing Scala programs using collections and provide improved performance.

5.3 OptiGraph

OptiGraph is a DSL for static graph analysis based on the Green-Marl DSL [3]. OptiGraph enables users to express graph analysis algorithms using graph-specific abstractions and automatically obtain efficient parallel execution. OptiGraph defines types for directed and undirected graphs, nodes, and edges. It allows data to be associated with graph nodes and edges via node and edge property types and provides three types of collections for node and edge storage (namely, `Set`, `Sequence`, and `Order`). Furthermore, OptiGraph defines constructs for `BFS` and `DFS` order graph traversal, sequential and explicitly parallel iteration, and implicitly parallel in-place reductions and group assignments. An important feature of OptiGraph is also its built-in support for bulk synchronous consistency via *deferred assignments*.

Figure 3 shows the parallel loop of the PageRank algorithm [16] written in both OptiGraph and Green-Marl. PageRank is a well-known algorithm that estimates the relative importance of each node in a graph (originally of web pages

```
1 val PR = NodeProperty[Double](G)
2 for (t <- G.Nodes) {
3   val rank = (1-d) / N + d*
4     Sum(t.InNbrs){w => PR(w)/w.OutDegree}
5   diff += abs(rank - PR(t))
6   PR <= (t,rank)
7 }
```

(a) OptiGraph: PageRank

```
1 N_P<Double> PR;
2 Foreach (t: G.Nodes) {
3   Double rank = (1-d) / N + d*
4     Sum(w:t.InNbrs){w.PR/w.OutDegree()};
5   diff += | rank - t.PR |;
6   t.PR <= rank @ t;
7 }
```

(b) Green-Marl: PageRank

Fig. 3. The major portion of the PageRank algorithm implemented in both OptiGraph and Green-Marl. OptiGraph is derived from Green-Marl, but required small syntactic changes in order to be embedded in Scala.

and hyperlinks) based on the number and page-rank of the nodes associated with its incoming edges (`InNbrs`). OptiGraph's syntax is slightly different since it is embedded in Scala and must be legal Scala syntax. However, the differences are small and the OptiGraph code is not more verbose than the Green-Marl version. In the snippet, `PR` is a node property associating a page-rank value with every node in the graph. The `<=` statement is a deferred assignment of the new page-rank value, `rank`, for node `t`; deferred writes to `PR` are made visible after the **`for`** loop completes via an explicit assignment statement (not shown). Similarly, `+=` is a scalar reduction that implicitly writes to `diff` only after the loop completes. In contrast, `Sum` is an in-place reduction over the parents of node `t`. This example shows that OptiGraph can concisely express useful graph algorithms in a naturally parallelizable way; the `ForeachReduce` Delite op implicitly injects the necessary synchronization into the **`for`** loop.

5.4 OptiMesh

OptiMesh is an implementation of Liszt on Delite. Liszt is a DSL for mesh-based partial differential equation (PDE) solvers [2]. Liszt code allows users to perform iterative computation over mesh elements (e.g. cells, faces). Data associated with mesh elements are stored in external fields that are indexed by the elements. Listing 1.3 shows a simple OptiMesh program that computes the flux through edges in the mesh. The **`for`** statement in OptiMesh is implicitly parallel and can only be used to iterate over mesh elements. `head` and `tail` are built-in accessors used to navigate the mesh in a structured way. In this snippet, the `Flux` field stores the flux value associated with a particular vertex. As the snippet demonstrates, a key challenge with OptiMesh is to detect write conflicts within for comprehensions given a particular mesh input.

OptiMesh solves this challenge by implementing the same domain-specific transformation as Liszt. First, the OptiMesh program is symbolically evaluated with a real mesh input to obtain a stencil of mesh accesses in the program. After the stencil is collected, an interference graph is built and disjoint loops are constructed using coloring. OptiMesh uses a Delite `Transformer` to simplify this implementation – the transformation is less than 100 lines of code. OptiMesh was the only new DSL we implemented that required a domain-specific transformation; the others were

```
1  for (edge <- edges(mesh)) {
2    val flux = flux_calc(edge)
3    val v0 = head(edge)
4    val v1 = tail(edge)
5    Flux(v0) += flux // possible write conflicts!
6    Flux(v1) -= flux
7  }
```

Listing 1.3. OptiMesh: simple flux computation

DSL	Delite Ops	Generic Opts.	Domain-Specific Opts.
OptiQL	Map, Reduce, Filter, Sort, Hash, Join	CSE, DCE, code motion, fusion, SoA, DFE	
OptiGraph	ForeachReduce, Map, Reduce, Filter	CSE, DCE, code motion, fusion	
OptiMesh	ForeachReduce	CSE, DCE, code motion	stencil collection & coloring transformation
OptiCollections	Map, Reduce, Filter, Sort, Hash, ZipWith, FlatMap	CSE, DCE, code motion, fusion, SoA, DFE	

Fig. 4. Sharing of DSL operations and optimizations

able to produce high performance results just by reusing generic optimizations and parallel code generation.

5.5 Reuse

By embedding the front-ends of our DSLs in Scala, we did not have to implement any lexing, parsing, or type checking. As we showed in the OptiGraph vs. Green-Marl example, the syntactic difference compared to a stand-alone DSL can still be relatively small. By embedding our DSL back-ends in Delite, each DSL was able to reuse parallel patterns, generic optimizations, common library functionality (e.g. math operators), and code generators for free. One important characteristic of the embedded approach is that when a feature (e.g. a parallel pattern) is added to support a new DSL, it can be reused by all subsequent DSLs. For example, we added the `ForeachReduce` pattern for OptiGraph, but it is also used in OptiMesh.

Figure 4 summarizes the characteristics and reuse of the new DSLs introduced in this section. The DSLs inherit most of their functionality from Delite, in the form of a small set of reused parallel patterns and generic optimizations. The DSLs use just 9 Delite ops total; seven ops (77.7%) were used in at least two DSLs; three (33.3%) were used in at least three DSLs. At the same time the DSLs are not constrained to built-in functionality, as demonstrated by OptiMesh's domain-specific optimizations.

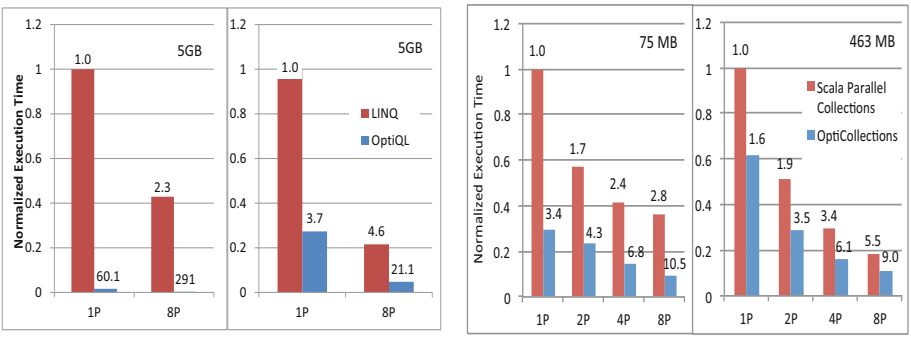
6 Case Studies

We present four case studies to evaluate our new DSLs. The first two case studies compare individual DSL performance against existing alternative programming

environments and the second two evaluate applications composing the DSLs in the two ways described in this paper (open and closed world). OptiQL is compared to LINQ [15] and OptiCollections to Scala Collections [17]. LINQ and Scala Collections are optimized libraries running on managed platforms (C#/CLR and Scala/JVM). We compare OptiMesh and OptiGraph to Liszt [2] and Green-Marl [3] respectively, stand-alone DSLs designed for high performance on heterogeneous hardware. Liszt and Green-Marl both generate and execute C++ code and have been shown to outperform hand-optimized C++ for the applications shown in this section. Each Delite DSL generated Scala code for the CPU and CUDA code for the GPU. For composability, we compare against an analogous Scala library implementation of each application when using a combination of DSLs to solve larger application problems.

All of our experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67GHz processors, 96GB of RAM, and an NVidia Tesla C2050. The CPU generated Scala code was executed on the Oracle Java SE Runtime Environment 1.7.0 and the Hotspot 64-bit server VM with default options. For the GPU, Delite executed CUDA v4.0. We ran each application ten times (to warm up the JIT) and report the average of the last 5 runs. For each run we timed the computational portion of the application. For each application we show normalized execution time relative to our DSL version with the speedup listed at the top of each bar.

6.1 Compiled Embedded vs. Optimized Library



(a) OptiQL: TPC-H Q1 (left) and Q2 (right) (b) OptiCollections: web-link benchmark

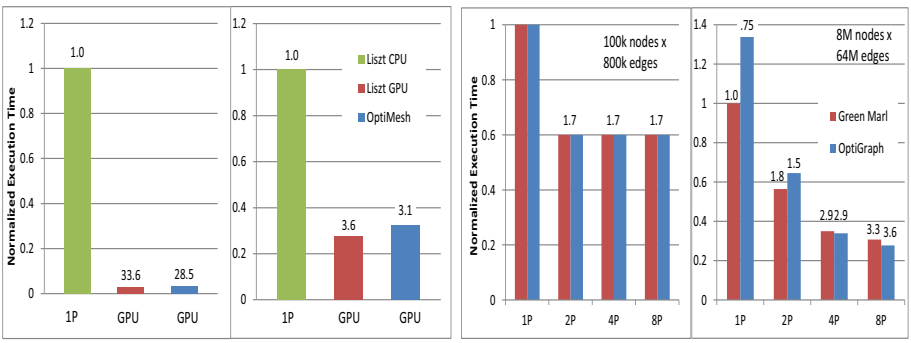
Fig. 5. Normalized execution time of applications written in OptiQL and OptiCollections. Speedup numbers are reported on top of each bar.

OptiQL: TPC-H Queries 1 & 2 Figure 5(a) compares the performance of queries 1 & 2 of the popular TPC-H benchmark suite on OptiQL vs. LINQ. Without any optimizations, OptiQL performance for the queries (not shown) is comparable to LINQ performance. However, such library implementations of the operations suffer from substantial performance penalties compared to an optimized and

compiled implementation. Naïvely, the query allocates a new table (collection) for each operation and every element of those tables is a heap-allocated object to represent the row. The two most powerful optimizations we perform are converting to an SoA representation and fusing across `GroupBy` operations to create a single (nested) loop to perform the query. Transforming to an SoA representation allows OptiQL to eliminate all object allocations for Q1 since all of the fields accessed by Q1 are JVM primitive types, resulting in a data layout consisting entirely of JVM primitive arrays. All together these optimizations provide 125x speedup over parallel LINQ for Q1 and 4.5x for Q2.

OptiCollections: Reverse web-link graph Figure 5(b) shows the result for the reverse web-link application discussed in Section 5.2 running on OptiCollections compared to Scala Parallel Collections. Scala Parallel Collections is implemented using Doug Lea's highly optimized fork/join pool with work stealing [18]. The OptiCollections version is significantly faster at all thread counts and scales better with larger datasets. The improvement is due to staged compilation, which helps in two ways. First, OptiCollections generates statically parallelized code. Unlike Scala collections, functions are inlined directly to avoid indirection. On the larger dataset, this does not matter as much, but on the smaller dataset the Scala collections implementation has higher overhead which results in worse scaling. Second, the OptiCollections implementation benefits from fusion and from transparently mapping `(Int,Int)` tuples to `Long`s in the back-end. These optimizations greatly reduce the number of heap allocations in the parallel operations, which improves both scalar performance and scalability.

6.2 Compiled Embedded vs. External DSL



(a) OptiMesh: Shallow Water (left) Scalar Convection (right)

(b) OptiGraph: PageRank

Fig. 6. Normalized execution time of applications written in OptiMesh and OptiGraph. Speedup numbers are reported on top of each bar.

Productivity. First, we consider the programming effort required to build OptiMesh and OptiGraph compared to the stand-alone versions they were based

on. Each Delite DSL compiler took approximately 3 months by a single graduate student to build using Delite. OptiMesh (≈5k loc) and OptiGraph (≈2.5k loc) were developed by 1st year Ph.D. students with no prior experience with Scala or Delite. In comparison, Liszt (≈25k loc Scala/C++) took a group of 2-3 compiler experts approximately one year to develop and Green-Marl (≈30k loc mostly C++) took a single expert approximately 6 months. As discussed in Section 5.5, most of the code reduction is due to reuse from being both front and back-end embedded (in Scala and Delite respectively). The Delite DSLs did not need to implement custom parsers, type checkers, base IRs, schedulers, or code generators. Similarly, while OptiMesh and OptiGraph do not implement all of the optimizations performed by Liszt and Green-Marl, they inherit other Delite optimizations (e.g. fusion) for free. For comparison, the Delite framework is ≈11k and LMS is ≈7k lines of Scala code. It is interesting to note that ≈4k lines of the Liszt code is for Pthreads and CUDA parallelization; a major benefit of using Delite is that parallelization for multiple targets is handled by the framework.

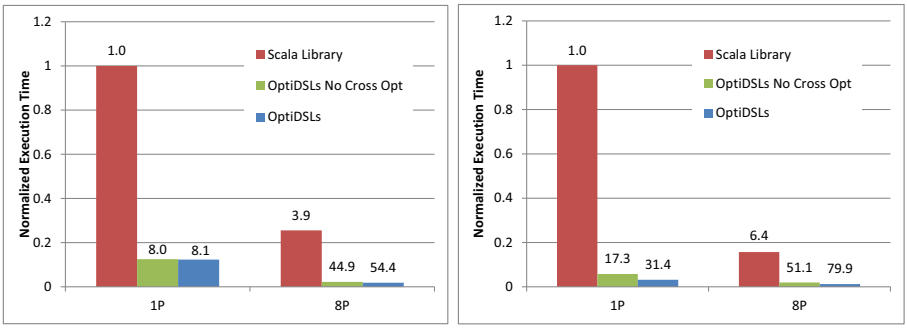
OptiMesh: Shallow water simulation & Scalar convection Figure 6(a) shows the performance of OptiMesh and Liszt on two scientific applications. Each application consists of a series of `ForeachReduce` operations surrounded by iterative control-flow to step the time variable of the PDE. It is well-suited to GPU execution as mesh sizes are typically large and the cost of copying the input data to the GPU is small compared to the amount of computation required. However, the original applications around which the Liszt language was designed were only implemented using MPI. Liszt added GPU code generation and demonstrated significant speedups compared to the CPU version. For both OptiMesh applications, Delite is able to generate and execute a CUDA kernel for each colored foreach loop and achieve performance comparable to the Liszt GPU implementation. Liszt's (and OptiMesh's) ability to generate both CPU and GPU implementations from a single application source illustrates the benefit of using DSLs as opposed to libraries that only target single platforms.

OptiGraph: PageRank Figure 6(b) compares the performance of the PageRank algorithm [16] implemented in OptiGraph to the Green-Marl implementation on two different uniform random graphs of sizes 100k nodes by 800k edges and 8M nodes by 64M edges, respectively. This benchmark is dominated by the random memory accesses during node neighborhood exploration. Since OptiGraph's memory access patterns and the memory layout of its back-end data structures are similar to those of Green-Marl, OptiGraph's sequential performance and scalability across multiple processors is close to that of Green-Marl. Although the smaller graph fits entirely in cache, the parallel performance is limited by cache conflicts when accessing neighbors and the associated coherency traffic. The sequential difference between OptiGraph and Green-Marl in the larger graph can be attributed to the difference between executing Scala generated code vs. C++. However, as we increase the number of the cores, the benchmark becomes increasingly memory-bound and the JVM overhead becomes negligible.

```
1  def valueIteration(actionResults: Rep[Map[Action, (Matrix[Double],Vector[Double])]],
2      initValue: Rep[Vector[Double]], discountFactor: Rep[Double],
3      tolerance: Rep[Double]) = {
4    val bestActions = Seq[Action](initValue.length)
5    var (value, delta) = (initValue, Double.MaxValue)
6    while (abs(delta) > tolerance) {
7      val newValue = Vector.fill(0,value.length) { i =>
8        val allValues = actionResults map { case (action,(prob,cost)) =>
9          (action, (prob(i) * value(i) * discountFactor + cost(i)).sum) }
10       val minActionValue = allValues reduce { case ((act1,v1),(act2,v2)) =>
11         if (v1 <= v2) (act1,v1) else (act2,v2) }
12       bestActions(i) = minActionValue.key; minActionValue.value }
13     delta = diff(newValue, value); value = newValue }
14   (value, bestActions) }
```

Listing 1.4. Value Iteration of a Markov Decision Process

6.3 Open-World Composability



(a) small action set (b) large action set

Fig. 7. Normalized execution time of value iteration of a Markov decision process. Performance is shown both with and without cross-DSL optimization. Speedup numbers are reported on top of each bar.

We illustrate open-world composability by implementing the value iteration algorithm for a Markov decision process (MDP), shown in Listing 1.4, using two different DSLs, OptiLA and OptiCollections. OptiLA is a DSL for linear algebra providing `Matrix` and `Vector` operations, and is specifically designed to be included by other DSLs by making no closed-world assumptions. OptiLA is a refactored portion of OptiML [11], a machine learning DSL designed for statistical inference problems expressed with vectors, matrices, and graphs. Although OptiML originally contained its own linear algebra components, we have found that several DSLs need some linear algebra capability, so we extracted OptiLA and modified OptiML to extend it using the techniques in Section 4.1. OptiCollections was also designed to be included by other DSLs and applications, as described in Section 5.2.

The algorithm uses an OptiCollections `Map` to associate each user-defined `Action` with a probability density `Matrix` and cost `Vector`. OptiLA operations (line 8) compute the value for the next time step based on an action, and OptiCollections operations apply the value propagation to every action and then find the minimal value over all actions. This process is repeated until the minimizing value converges.

Figure 7 shows the performance for this application implemented using Scala Parallel Collections compared to using OptiLA and OptiCollections. For both datasets, the OptiDSL version shows significant speedup over the library implementation as well as improved parallel performance, due mainly to two key optimizations: loop fusion and AoS to SoA transformation. The latter transformation eliminates all of the tuples and class wrappers in lines 7-10, leaving only nested arrays. The OptiDSLs "No Cross Opt" bar simulates the behavior of compiling the code snippets for each DSL independently and then combining the resulting optimized code snippets together using some form of foreign function interface. Therefore this version does not include SoA transformations or fusion across DSLs, but does still fuse operations fully contained within a single DSL, most notably the OptiLA code snippet on line 8 of Listing 1.4. Figure 7(a) shows only very modest speedups after adding DSL cross-optimization. This is because the majority of the execution time is spent within the OptiLA code snippet, and so only fusion within OptiLA was necessary to maximize performance. Figure 7(b), however, shows the behavior for different data dimensions. In this case the total execution time spent within the OptiLA section is small, making fusion across the nested OptiCollections/OptiLA operations critical to maximizing performance.

Overall this case study shows that while sometimes applications can achieve good performance by simply offloading key pieces of computation to a highly optimized implementation (applications that call BLAS libraries are a classic example of this), other applications and even the same application with a different dataset require the compiler to reason about all of the pieces of computation together at a high level and be capable of performing optimizations across them in order to maximize performance.

6.4 Closed-World Composability

In this example, we combine OptiQL, OptiGraph, and OptiML using the closed-world composition strategy discussed in Section 4.2. We use the three DSLs together to implement a data analysis application on a Twitter dataset used in [19]. A truncated version of the application code is shown in Listing 1.5. The idea is to compute statistics related to the distribution of all tweets, of retweets (tweets that have been repeated by another user), and of the relationship between user connectivity and the number of times they have been retweeted.

The application follows a typical data analytic pipeline. It first loads data from a log file containing tweets with several attributes (sender, date, text, etc.). It then queries the dataset to extract relevant information using an OptiQL `Where` statement. The filtered data is passed on to OptiGraph and OptiML which both

```
type Tweet = Record { val time: String; val fromId: Int; val toId: Int;
                      val lang: String; val text: String; val RT: Boolean }
val q = OptiQL {
  //tweets: Table[Tweet]
  val reTweets = tweets.Where(t => t.lang == "en"
                  && Date(t.time) > Date("2008-01-01")  && t.RT)
  val allTweets = tweets.Where(t => t.lang == "en")
  DRef((reTweets.toArray, allTweets.toArray))
}
val g = OptiGraph {
  val in = q.get
  val G = Graph.fromArray(in._1.map(t => (t.fromId, t.toId)))
  val (LCC, RT) = (NodeProperty[Double](G), NodeProperty[Double](G))
  Foreach(G.Nodes) { s =>
    // count number of nodes connected in a triangle (omitted)
    if (total < 1) LCC(s) = 0 else LCC(s) = triangles.toDouble / total
    RT(s) = G.InNbrs(s).length
  }
  DRef((LCC.toArray, RT.toArray, in._1, in._2))
}
val r = OptiML {
  val in = g.get
  val scaledRT = norm(log((Vector.fromArray(in._2) + 1)))
  val X = Matrix(Vector.ones(in._1.length), Vector.fromArray(in._1))
  val theta = (X*X.t).inv * (X*scaledRT)  // unweighted linear regression
  // compute statistics on tweets (omitted)
}
```

Listing 1.5. Twitter Graph Analysis using multiple DSLs

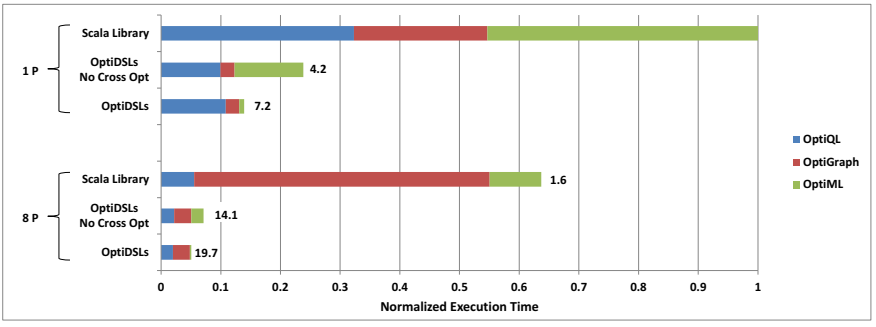


Fig. 8. Normalized execution time of Twitter data analysis. Performance is shown for each DSL section with and without cross-DSL optimization. Speedup numbers are reported on top of each bar.

analyze it and compute statistics. OptiGraph builds a graph where nodes are users and edges are retweets and computes the LCC (local clustering coefficient) and retweet counts for each user. The LCC is a loose approximation of the importance of a particular user. The OptiML section fits the filtered tweet data from OptiQL to a normal distribution and also runs a simple unweighted linear regression on the LCC and retweet counts that the OptiGraph code computed. The final results of the application are the distribution and regression coefficients.

Figure 8 shows the performance of this application implemented with Scala Parallel Collections compared to the Delite DSLs with and without cross-DSL optimization. The graph is broken down by the time spent in each DSL section (or for the library version, in the corresponding Scala code). Since we are using the closed world model, each DSL is independently staged, lowered to the common Delite representation, and re-compiled. The application code is still a single program and the DSLs pass data in memory without any boxing overhead. Therefore, in contrast to using stand-alone compiled DSLs for each computation, we incur no serialization overhead to pipe data from one DSL snippet to the other. The Scala library version, which is also a single application, is approximately 5x slower than the non-cross-optimized DSL version on 1 thread and 10x slower on 8 threads. The speedup is due to optimizations performed by Delite across all DSLs. The OptiQL code benefits from filter fusion as well as lifting the construction of a `Date` object outside of the filter loop, which demonstrates the high-level code motion that is possible with more semantic information (the `Date` comparison is a virtual method call for Scala, so it does not understand that the object is constant in the loop). The OptiGraph version is faster than the corresponding library snippet mainly due to compiling away abstractions (we use only primitive operations on arrays in the generated code compared to Scala's `ArrayBuffer`, which has run-time overhead). The OptiGraph code is also the least scalable, since this particular graph is highly skewed to a few dominant nodes and the graph traversal becomes very irregular. Due to the additional overhead of the library version, this effect is more pronounced there. Finally, the OptiML version is faster mainly because of loop fusion and CSE across multiple linear algebra operations.

Co-optimizing the DSLs, which is enabled by composing them, produces further opportunities. The Delite compiler recognizes that OptiGraph and OptiML together require only 4 fields per tweet of the original 6 (OptiGraph uses "toId" and "fromId" and OptiML uses "text" and "hour"). The remaining fields are DFE'd by performing SoA transformation on the filter output and eliminating arrays that are not consumed later. The other major cross DSL optimization we perform is to fuse the filter from OptiQL with their consumers in OptiGraph and OptiML. Note that the fusion algorithm is strictly data dependent; the OptiML snippet and OptiQL snippets are syntactically far apart, but can still be fused. This example also shows that since the DSL blocks are composed into a single IR, we can fuse across multiple scope boundaries when co-optimizing. All together, cross optimizations result in an extra 1.72x sequential speedup over the composed Delite DSL version without cross optimizations.

7 Related Work

Our embedded DSL compilers build upon numerous previously published work in the areas of DSLs, extensible compilers, heterogeneous compilation, and parallel programming.

There is a rich history of DSL compilation in both the embedded and stand-alone contexts. Elliot et al. [4] pioneered embedded compilation and used a simple image synthesis DSL as an example. Feldspar [20] is an embedded DSL that combines shallow and deep embedding of domain operations to generate high performance code. For stand-alone DSL compilers, there has been considerable progress in the development of parallel and heterogeneous DSLs. Liszt [2] and Green-Marl [3] are discussed in detail in this paper. Diderot [21] is a parallel DSL for image analysis that demonstrates good performance compared to hand-written C code using an optimized library. The Spiral system [22] progressively lowers linear transform algorithms through a series of different DSLs to apply optimizations on different levels [23]. Subsets of Spiral have also been implemented using Scala and LMS. Giarrusso et al. [24] investigate database-like query optimizations for collection classes and present SQuOpt, a query optimizer for a DSL that, like OptiCollections, mimics the Scala collections API and also uses techniques similar to LMS to obtain an IR for relevant program expressions. Our work aggregates many of the lessons and techniques from previous DSL efforts and makes them easier to apply to new domains, and to the problem of composing DSLs.

Recent work has begun to explore how to compose domain-specific languages and runtimes. Mélusine [25] uses formal modeling to define DSLs and their composition. Their approach attempts to reuse existing models and their mappings to implementations. Dinkelar et al. [26] present an architecture for composing purely embedded DSLs using aspect-oriented concepts; a meta-object is shared between all the eDSLs and implements composition semantics such as join points. MadLINQ [27] is an embedded matrix DSL that integrates with DryadLINQ [28], using LINQ as the common back-end. Compared to these previous approaches, our work is the first to demonstrate composition and co-optimization with high performance, statically compiled DSLs.

There has also been work on extensible compilation frameworks aimed towards making DSLs and high performance languages easier to build. Racket [29] is a dialect of Scheme designed to make constructing new programming languages easier. Spoofax [30] and JetBrains MPS [31] are language workbenches for defining new DSLs and can generate automatic IDE support from a DSL grammar. While these efforts also support DSL reuse and program transformation, they are generally more focused on expressive DSL front-ends, whereas Delite's emphasis is on high performance and heterogeneous compilation. Both areas are important to making DSL development easier and could be used together to complement each other. On the performance side, telescoping languages [32] automatically generate optimized domain-specific libraries. They share Delite's goal of incorporating domain-specific knowledge in compiler transformations. Delite compilers

extend optimization to DSL data structures and also optimize both the DSL and the program using it in a single step.

Outside the context of DSLs, there have been efforts to compile high-level general purpose languages to lower-level (usually device-specific) programming models. Mainland et al. [6] use type-directed techniques to compile an embedded array language, Nikola, from Haskell to CUDA. This approach suffers from the inability to overload some of Haskell's syntax (if-then-else expressions) which is not an issue with our version of the Scala compiler. Copperhead [7] automatically generates and executes CUDA code on a GPU from a data-parallel subset of Python. Nystrom et al. [33] show a library-based approach to translating Scala programs to OpenCL code through Java bytecode translation. Since the starting point of these compilers is byte code or generic Python/Java statements, the opportunities for high-level optimizations are more limited relative to DSL code.

8 Conclusion

In this paper we showed that a common back-end can be used to compose high performance, compiled domain-specific languages. The common back-end also provides a means to achieve meaningful reuse in the compiler implementations when targeting heterogeneous devices. We demonstrated this principle by implementing four new diverse DSLs (OptiQL, OptiCollections, OptiGraph, and OptiMesh) in Delite, an extensible compilation framework for compiled embedded DSLs. The DSLs required only 9 parallel operators and 7 were reused in at least two DSLs. We showed that OptiQL and OptiCollections exceed the performance of optimized library implementations by up to 125x. OptiGraph and OptiMesh are both based on existing stand-alone DSLs (Green-Marl and Liszt respectively) but require less code to build and achieve no worse than 30% slow-down. In addition to each DSL providing high performance and targeting multicore and GPU architectures, applications composing multiple DSLs perform well and benefit from cross-DSL optimization. To the best of our knowledge, this work is the first to demonstrate high performance compiled DSL composability.

Acknowledgements. We are grateful to the anonymous reviewers for their detailed suggestions, to Nada Amin for her assistance with the Scala-Virtualized compiler, and to Peter Kessler and Zach DeVito for reviewing previous versions of this paper. This research was sponsored by DARPA Contract, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army contract AHPCRC W911NF-07-2-0027-1; NSF grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore, IIS-1247701; NSF grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, and NVIDIA; and European Research Council (ERC) under grant 587327 "DOPPLER". Authors also acknowledge additional support from Oracle. The views and conclusions contained herein are those of the authors

and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. In: PPoPP (2011)
2. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: A domain specific language for building portable mesh-based PDE solvers. In: SC (2011)
3. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-marl: A DSL for easy and efficient graph analysis. In: ASPLOS (2012)
4. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. In: Taha, W. (ed.) SAIG 2000. LNCS, vol. 1924, pp. 9–26. Springer, Heidelberg (2000)
5. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: DSL, pp. 109–122. ACM, New York (1999)
6. Mainland, G., Morrisett, G.: Nikola: embedding compiled GPU functions in haskell. In: Haskell 2010, pp. 67–78. ACM, New York (2010)
7. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: compiling an embedded data parallel language. In: PPoPP, pp. 47–56. ACM, New York (2011)
8. Brown, K.J., Sujeeth, A.K., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. In: PACT (2011)
9. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In: GPCE, pp. 127–136. ACM, New York (2010)
10. Rompf, T., Sujeeth, A.K., Lee, H., Brown, K.J., Chafi, H., Odersky, M., Olukotun, K.: Building-blocks for performance oriented DSLs. In: DSL (2011)
11. 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: ICML (2011)
12. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. 248(1-2), 211–242 (2000)
13. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. In: PEPM (2012)
14. Rompf, T., Sujeeth, A.K., Amin, N., Brown, K., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., Odersky, M.: Optimizing data structures in high-level programs. In: POPL (2013)
15. Meijer, E., Beckman, B., Bierman, G.: LINQ: Reconciling object, relations and XML in the .NET framework. In: SIGMOD, p. 706. ACM, New York (2006)
16. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (November 1999), Previous number = SIDL-WP-1999-0120
17. Prokopec, A., Bagwell, P., Rompf, T., Odersky, M.: A generic parallel collection framework. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 136–147. Springer, Heidelberg (2011)
18. Lea, D.: A java fork/join framework. In: JAVA 2000, pp. 36–43. ACM, New York (2000)

19. Yang, J., Leskovec, J.: Patterns of temporal variation in online media. In: WSDM 2011, pp. 177–186. ACM, New York (2011)
20. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of feldspar: An embedded language for digital signal processing. In: Hage, J., Morazán, M.T. (eds.) IFL 2010. LNCS, vol. 6647, pp. 121–136. Springer, Heidelberg (2011)
21. Chiw, C., Kindlmann, G., Reppy, J., Samuels, L., Seltzer, N.: Diderot: a parallel DSL for image analysis and visualization. In: PLDI, pp. 111–120. ACM, New York (2012)
22. Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R., Rizzolo, N.: Spiral: Code generation for DSP transforms. Proceedings of the IEEE 93(2), 232–275 (2005)
23. Franchetti, F., Voronenko, Y., Püschel, M.: Formal loop merging for signal transforms. In: PLDI, pp. 315–326 (2005)
24. Giarrusso, P.G., Ostermann, K., Eichberg, M., Mitschke, R., Rendel, T., Kästner, C.: Reify your collection queries for modularity and speed! In: AOSD (2013)
25. Estublier, J., Vega, G., Ionita, A.D.: Composing domain-specific languages for wide-scope software engineering applications. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 69–83. Springer, Heidelberg (2005)
26. Dinkelaker, T., Eichberg, M., Mezini, M.: An architecture for composing embedded domain-specific languages. In: AOSD, pp. 49–60. ACM (2010)
27. Qian, Z., Chen, X., Kang, N., Chen, M., Yu, Y., Moscibroda, T., Zhang, Z.: Madlinq: large-scale distributed matrix computation for the cloud. In: EuroSys 2012, pp. 197–210. ACM, New York (2012)
28. Isard, M., Yu, Y.: Distributed data-parallel computing using a high-level programming language. In: SIGMOD, pp. 987–994. ACM, New York (2009)
29. Flatt, M.: Creating languages in racket. Commun. ACM 55(1), 48–56 (2012)
30. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA 2010, pp. 444–463. ACM, New York (2010)
31. JetBrains: Meta Programming System (2009)
32. Kennedy, K., Broom, B., Chauhan, A., Fowler, R., Garvin, J., Koelbel, C., McCosh, C., Mellor-Crummey, J.: Telescoping languages: A system for automatic generation of domain languages. Proceedings of the IEEE 93(3), 387–408 (2005)
33. Nystrom, N., White, D., Das, K.: Firepile: run-time compilation for GPUs in scala. In: GPCE, pp. 107–116. ACM, New York (2011)

Combining Form and Function: Static Types for JQuery Programs*

Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi

Brown University

Abstract. The jQuery library defines a powerful *query language* for web applications' scripts to interact with Web page content. This language is exposed as jQuery's API, which is implemented to fail silently so that incorrect queries will not cause the program to halt. Since the correctness of a query depends on the structure of a page, discrepancies between the page's actual structure and what the query expects will also result in failure, but with no error traces to indicate where the mismatch occurred.

This work proposes a novel type system to statically detect jQuery errors. The type system extends Typed JavaScript with *local structure* about the page and with *multiplicities* about the structure of containers. Together, these two extensions allow us to track precisely which nodes are active in a jQuery object, with minimal programmer annotation effort. We evaluate this work by applying it to sample real-world jQuery programs.

1 Introduction

Client-side web applications are written in JavaScript (JS), using a rich, but low-level, API known as the Document Object Model (DOM) to manipulate web content. Essentially an "assembly language for trees", these manipulations consist of selecting portions of the document and adding to, removing, or modifying them. Like assembly language, though, programming against this model is imperative, tedious and error-prone, so web developers have created JS libraries that abstract these low-level actions into higher-level APIs, which essentially form their own domain-specific language.

We take as a canonical example of this effort the *query portion* of the popular jQuery library, whose heavily stylized form hides the imperative DOM plumbing under a language of "sequences of tree queries and updates". This query-language abstraction is widely used: other JS libraries, such as Dojo, Yahoo!'s YUI, Ember.js and D3, either embed jQuery outright or include similar query APIs. (All these libraries also include various features and functionality not related to document querying; these features differ widely between libraries, and are not our focus here.) JQuery in particular enjoys widespread adoption, being used in over half of the top hundred thousand websites [4].

* This work is partially supported by the US National Science Foundation and Google.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 79–103, 2013.

From a program-analysis standpoint, jQuery can also drastically simplify the analysis of these programs, by directly exposing their high-level structure. Indeed, "jQuery programs" are written so dramatically differently than "raw DOM programs" that in this work, we advocate understanding jQuery as a language in its own right. Such an approach lets us address two essential questions. First, what bugs are idiomatic to this high-level query-and-manipulation language, and how might we detect them? And second, if a jQuery-like language were designed with formal analyses in mind, what design choices might be made differently?

Contributions. This work examines *query errors*, where the result of a query returns too many, too few, or simply undesired nodes of the document. We build a type system to capture the behavior of jQuery's document-query and manipulation APIs; our goal is to expose all query errors as type errors. (Our type system is not *jQuery*-specific, however, and could be applied to any of the query languages defined by JS libraries.) In designing this type system, we find a fundamental tension between the flexibility of jQuery's APIs and the precision of analysis, and propose a small number of new APIs to improve that loss of precision.

At a technical level, this paper extends prior work [10, 16, 17] that has built a sophisticated type system for JS, to build a domain-specific *type* language for analyzing jQuery. Our type system enhances the original with two novel features:

- *Multiplicities*, a lightweight form of indexed types [26] for approximating the sizes of container objects; and
- *Local structure*, a lightweight way to inform the type system of the "shape" of relevant portions of the page, obviating the need for a global page schema.

Combined, these two features provide a lightweight dependent type system that allows us to typecheck sophisticated uses of jQuery APIs accurately and precisely, with minimal programmer annotation. Our prototype implementation is available at `http://www.jswebtools.org/jQuery/`.[1]

We introduce a running example (Section 2) to explain the key ideas of our approach (Sections 3 and 4), and highlight the key challenges in modeling a language as rich as jQuery (Sections 5 and 6). We evaluate its flexibility using several real-world examples (Section 7), and describe related work and directions for future improvements (Sections 8 and 9).

2 Features and Pitfalls of the JQuery Language

JQuery's query APIs break down into three broad groups: *selecting* an initial set of nodes in the page, *navigating* to new nodes in the document relative to those nodes, and *manipulating* the nodes. These functions allow the programmer to process many similar or related nodes easily and uniformly. To illustrate most of

[1] We were unable to submit this artifact for evaluation per that committee's rules, because our fourth author co-chaired that committee.

these APIs and their potential pitfalls, and as a running example, we examine a highly simplified Twitter stream. A stream is simply a ⟨**div**/⟩ element containing tweets. Each tweet in turn is a ⟨**div**/⟩ element containing a timestamp, an author, and the tweet content:

⟨**div** class=“stream”⟩
  ⟨**div** class=“tweet”⟩
    ⟨**p** class=“time”/⟩ ⟨**p** class=“author”/⟩ ⟨**p** class=“content short”/⟩
  ⟨/**div**⟩
  ...
⟨/**div**⟩

Selecting Nodes. The jQuery `$` function is the entry point for the entire API. This function typically takes a `CSS` selector string and returns a jQuery object containing *all* the elements in the document that match the provided selector:

```
1 $("*.time").css('color', 'red'); // Colors all timestamp elements
```

But the power and flexibility of this method may lead to silent errors. *How do we assure ourselves, for example, that `$("*.time")` matches any elements at all?*

Navigating Between Nodes. JQuery supplies several methods for relative movement among the currently selected nodes. These methods operate uniformly on all the nodes contained in the current collection:

```
1  // Returns a collection of the children of aTweet:
2  //   a timestamp, an author, and a content node
3  $(aTweet).children();
4  // Returns all the children of all tweets
5  $(".tweet").children();
6  // Returns the authors and contents of all tweets
7  $(".tweet").children().next();
8  // Returns the empty collection
9  $(".tweet").children().next().next().next().next();
10 // Meant to colorize contents, but has no effect
11 $(".tweet").children().next().next().next().css("color", "red");
```

The first two examples are straightforward: requesting the `children()` of a node returns a collection of all its children, while calling `children()` on several nodes returns all their children. The next example (line 7) shows jQuery's silent error handling: calling `next()` on a collection of timestamps, authors and contents will return a collection of the next siblings of each node, even if some nodes (here, content nodes) have no next siblings. In fact, jQuery does not raise an error even when calling a method on an empty collection: on line 9, after the third call to `next()`, there are no remaining elements for the final call. The final example highlights why this might be a problem: The programmer intended to select the contents nodes, but called `next()` once too often. The call to `css()` then dutifully changed the styles of all the nodes it was given—i.e., none—and returned successfully. This is another important error condition: *how can we assure ourselves that we haven't “fallen off the end” of our expected content?*

Manipulating Content. The purpose of obtaining a collection of nodes is to manipulate them, via methods such as `css()`. These manipulator functions all follow a common getter/setter pattern: for example, when called with one argument, `css()` returns the requested style property of the given node. When supplied with two arguments, it sets the style property of the given nodes.

Note carefully the different pluralities of getters and setters. Getters implicitly apply only to the *first* node of the collection, while setters implicitly apply to *all* the nodes of the collection. This can easily lead to errors: while in many cases, the order of elements in a collection matches the document order, and while in many other cases, the collection only contains a single element, neither of these conditions is guaranteed. *How can we assure ourselves that the collection contains exactly the node we expect as its "first" child?*

JQuery provides a general-purpose `each()` method, that allows the programmer to map a function over every node in the collection. (This is useful when the necessary computation requires additional state that cannot be expressed as a simple sequence of jQuery API calls.) This API is one of the few places where it is possible to trigger a run-time error while a jQuery function is on the call stack, precisely because the programmer-supplied function is *not* jQuery code. *How can we assure ourselves that the function calls only those methods on the element that are known to be defined?*

Chaining. JQuery's API revolves around the *chaining* pattern, where practically every method (that is not a getter) is designed to return a jQuery object: manipulation APIs return their invocant, and navigation APIs return a new collection of the relevant nodes. This allows for fluid idioms such as the following, simplified code that animates the appearance of a new tweet on Twitter (the precise arguments to these functions are not relevant; the "style" of API calls is):

```
1 $(aTweet).fadeIn(...).css(...).animate(...);
```

Each call returns its invocant object, so that subsequent calls can further manipulate it. Modeling this idiom precisely is crucial to a useful, usable type system.

Choosing a Type System. Though jQuery dynamically stifles errors, in this paper we argue that query errors are latent and important flaws that should be detected and corrected—statically, whenever possible. Accordingly, we aim to develop a type system that can capture the behaviors described above, and that can catch query errors without much programmer overhead. In particular, we would like type inference to work well, as any non-inferred types must be grudgingly provided by the programmer instead.

Because the behavior of a query depends crucially on the query parameters, the navigational steps, and any manipulations made, it would seem reasonable that the type of the query must depend on these values too, leading to a dependent type system whose types include strings and numbers to encode the query and its size. Unfortunately, in general, type inference for dependent type systems is undecidable, leading to large annotation or proof burdens on the programmer.

However, as we will see below, a type system tailored for jQuery can forego much of the complexity of general-purpose dependent types. By carefully restricting our use of strings, and by approximating our use of numbers, we can regain a "lightweight" dependently typed system with sufficient precision for our purposes, and that still enjoys decidable type inference in practice.

Comparisons with Other Query Languages. Describing jQuery as "a query language" invites comparison with other tree-query languages, most notably XPath and XQuery [23, 24], and the programming languages XDuce [11] and ℂDuce [2]. We compare our approach with XDuce and ℂDuce more thoroughly in Section 8; for now, we note that unlike `XML` databases that conform to schemas and from which strongly-typed data can be extracted, `HTML` pages are essentially schema-free and fundamentally presentational. JQuery therefore adopts `CSS` selectors, the language used to style web pages, as the basis of its query language.

3 Multiplicities

In this and the following section, we describe the novel features of our type system, and build up the central type definition in stages. Our system is an extension of Typed JavaScript [10], which provides a rich type system including objects types respecting prototype inheritance, (ordered) intersection and (unordered) union types, equi-recursive types, type operators and bounded polymorphism. The type language, including our new constructs, is shown in Fig. 1; we will introduce features from this type system as needed in our examples below.

Each of the three phases of the jQuery API described above induces a characteristic problem: in reverse order, ensuring that

1. Callback code only calls appropriate methods on the provided elements,
2. Precisely the intended target element is the first element of the jQuery object,
3. Navigation does not overshoot the end of known content, and
4. Selectors return the intended elements from the page.

Warmup. The first challenge is a traditional type-checking problem, and one that is well handled by existing Typed JavaScript. We simply must ensure that the supplied callback function has type `SomeElementType -> Undef`, provided we know which `SomeElementType` is expected by the current jQuery object. This leads to a (very simplistic) first attempt at "the jQuery type":

```
1 type jQuery = μ jq :: * => * .
2   Λ e :: *  . {
3     each : ['jq⟨'e⟩] ('e -> Undef) -> 'jq⟨'e⟩
4   }
```

$\alpha \in$ *Type and multiplicity variables*
$\kappa \in$ *Kind* $::= * \mid \texttt{M}\langle * \rangle$
$r \in$ *RegEx* $::=$ *regular expressions*
$\tau \in$ *Type* $::= \alpha \mid \texttt{Num} \mid r \mid \texttt{True} \mid \texttt{False} \mid \texttt{Undef} \mid \texttt{Null} \mid \top \mid \bot$
$\mid \texttt{ref}\ \tau \mid [\tau]\tau \times \cdots \to \tau \mid \tau\texttt{+}\tau \mid \tau\&\tau \mid \mu\alpha.\tau \mid \{\star : \tau, s : \tau, \ldots\}$
$\mid \forall\alpha \leq \tau.\tau \mid \forall\alpha :: \kappa.\tau \mid \Lambda\ \alpha :: \kappa.\tau \mid \tau\langle\tau\rangle \mid \tau\langle m\rangle$
$\mid \tau$ `@` *CSS selector*
$m \in$ *Mult* $::= \texttt{M}\langle\tau\rangle \mid \texttt{0}\langle\texttt{m}\rangle \mid \texttt{1}\langle\texttt{m}\rangle \mid \texttt{01}\langle\texttt{m}\rangle \mid \texttt{1+}\langle\texttt{m}\rangle \mid \texttt{0+}\langle\texttt{m}\rangle \mid m_1 +\!\!+ m_2$
$l \in$ *LS* $::= (\langle$*Name*$\rangle$ `:` $\langle$*ElementType*$\rangle$
`classes=[...] optional classes=[...] ids=[...]`
l`...)`
$\mid$ <*Name*> $\mid$ <*Name*>`+` $\mid$ `...`

Fig. 1. Syntax of types, multiplicities and local structure

In words, this defines `jQuery` to be a type constructor (of kind `* => *`, i.e., a function from types to types) that takes an element type (line 2) and returns an object with a field `each` (line 3), which is a function that must be invoked on a `jQuery` object containing elements of the appropriate type (the type in brackets) and passed a single callback argument (in parentheses) and returns the original jQuery object. So that the type of `each` can refer to the overall jQuery type, we say that `jQuery` is in fact a recursive type (line 1). The `'e` type parameter records the type of the elements currently wrapped by the jQuery object.

The Need for More Precision. The next two challenges cannot be expressed by traditional types. We need to keep track of *how many* elements are present in the current collection, so that we know whether calling a getter function like `css()` is ambiguous, or whether calling a navigation function like `next()` has run out of elements. (We defer the challenge of knowing exactly what type comes `next()` after a given one until Section 4; here we just track quantities.)

3.1 Defining Multiplicities

JQuery's APIs distinguish between zero, one and multiple items. To encode this information, we introduce a new kind that we call *multiplicities*, written $\texttt{M}\langle * \rangle$, with the following constructors:

$m \in$ *Mult* $::= \texttt{M}\langle\tau\rangle \mid \texttt{0}\langle\texttt{m}\rangle \mid \texttt{1}\langle\texttt{m}\rangle \mid \texttt{01}\langle\texttt{m}\rangle \mid \texttt{1+}\langle\texttt{m}\rangle \mid \texttt{0+}\langle\texttt{m}\rangle \mid m_1 +\!\!+ m_2$

The first of these constructors (`M`) embeds a single type into a multiplicity; for brevity, we often elide this constructor below. The next five assert the presence of zero, one, zero or one, one or more, or zero or more multiplicities. These multiplicities can be nested, but can be normalized by simple multiplication: for

$\cdot\langle\cdot\rangle$	$\texttt{0}\langle\tau\rangle$	$\texttt{1}\langle\tau\rangle$	$\texttt{01}\langle\tau\rangle$	$\texttt{1+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$
0	$\texttt{0}\langle\tau\rangle$	$\texttt{0}\langle\tau\rangle$	$\texttt{0}\langle\tau\rangle$	$\texttt{0}\langle\tau\rangle$	$\texttt{0}\langle\tau\rangle$
1	$\texttt{0}\langle\tau\rangle$	$\texttt{1}\langle\tau\rangle$	$\texttt{01}\langle\tau\rangle$	$\texttt{1+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$
01	$\texttt{0}\langle\tau\rangle$	$\texttt{01}\langle\tau\rangle$	$\texttt{01}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$
1+	$\texttt{0}\langle\tau\rangle$	$\texttt{1+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$	$\texttt{1+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$
0+	$\texttt{0}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$	$\texttt{0+}\langle\tau\rangle$

(a) Normalization for simple multiplicities

$\lfloor m +\!\!+ n \rfloor$	$\texttt{0}\langle\tau_1\rangle$	$\texttt{1}\langle\tau_1\rangle$	$\texttt{01}\langle\tau_1\rangle$	$\texttt{1+}\langle\tau_1\rangle$	$\texttt{0+}\langle\tau_1\rangle$
$\texttt{0}\langle\tau_2\rangle$	$\texttt{0}\langle\top\rangle$	$\texttt{1}\langle\tau_1\rangle$	$\texttt{01}\langle\tau_1\rangle$	$\texttt{1+}\langle\tau_1\rangle$	$\texttt{0+}\langle\tau_1\rangle$
$\texttt{1}\langle\tau_2\rangle$	$\texttt{1}\langle\tau_2\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$
$\texttt{01}\langle\tau_2\rangle$	$\texttt{01}\langle\tau_2\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{0+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{0+}\langle\tau_3\rangle$
$\texttt{1+}\langle\tau_2\rangle$	$\texttt{1+}\langle\tau_2\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$
$\texttt{0+}\langle\tau_2\rangle$	$\texttt{0+}\langle\tau_2\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{0+}\langle\tau_3\rangle$	$\texttt{1+}\langle\tau_3\rangle$	$\texttt{0+}\langle\tau_3\rangle$

(b) Simplification of sum multiplicities, where $\tau_3 = \tau_1 \texttt{+} \tau_2$, and m and n are normalized

Fig. 2. Normalization and simplification for multiplicities: simplifying sums assumes that the arguments have first been normalized

instance, $\texttt{01}\langle\texttt{1+}\langle\tau\rangle\rangle = \texttt{0+}\langle\tau\rangle$. (In words, zero or one groups of one or more τ is equal to zero or more τ.) The normalization table is listed in Fig. 2a.

The remaining multiplicity, the sum $m_1 +\!\!+ m_2$, asserts the presence of both m_1 and m_2, and is useful for concatenating two jQuery collections. Sum multiplicities do not normalize multiplicatively the way the previous ones do, but they can be approximated additively: for example, $\texttt{1}\langle\tau_1\rangle +\!\!+ \texttt{01}\langle\tau_2\rangle <: \texttt{1+}\langle\tau_1\texttt{+}\tau_2\rangle$. In words, if we have one τ_1 and zero or one τ_2, then we have one or more τ_1s or τ_2s (this union type is denoted by the plus symbol); the full rules are in Fig. 2b. This loses precision: the latter multiplicity also describes a collection of several τ_2s and zero τ_1s, while the original sum does not. Our type-checking algorithm therefore avoids simplifying sums whenever possible.

Note that while types describe expressions or values, multiplicities do not directly describe anything. Instead, they are permitted solely as arguments to type functions:[2] they are a lightweight way to annotate container types with a few key pieces of size information. In particular, they are easier to use than typical dependent types with arbitrary inequality constraints, as they can be manipulated syntactically, without needing an arithmetic solver.

We can now use these multiplicities to refine our definition for `jQuery`:

```
1 type jQuery = μ jq :: M⟨*⟩ => * .
2   Λ m :: M⟨*⟩  . {
3     each : ∀ e <: Element .
4            ['jq⟨0+⟨'e⟩⟩] ('e -> Undef) -> 'jq⟨'m⟩
5     css : ∀ e <: Element .
6           (['jq⟨1⟨'e⟩⟩] Str -> Str &
7            ['jq⟨1+⟨'e⟩⟩] Str*Str -> 'jq⟨1+⟨'e⟩⟩)
8   }
```

The kind for `jq` has changed to accept a multiplicity, rather than a bare type. The type for `each` has also changed to recover the type `e` describing elements, which we bound above by `Element`. Multiplicities also give us sufficient precision

[2] This is exactly analogous to the distinction between types and ordinals in indexed types [1, 26], or between types and sequence types in XDuce [11]; see Section 8.

m_1 <: m_2	0⟨τ⟩	1⟨τ⟩	01⟨τ⟩	1+⟨τ⟩	0+⟨τ⟩
0⟨τ⟩	✓*		✓		✓
1⟨τ⟩		✓	✓	✓	✓
01⟨τ⟩			✓		✓
1+⟨τ⟩				✓	✓
0+⟨τ⟩					✓

$$\text{M-Typ}\ \frac{\tau <: \tau'}{\mathtt{M}\langle\tau\rangle <: \mathtt{M}\langle\tau'\rangle} \qquad \text{M-Sums}\ \frac{(m_1 <: m_3 \wedge m_2 <: m_4) \vee (m_1 <: m_4 \wedge m_2 <: m_3)}{m_1 + m_2 <: m_3 + m_4}$$

$$\text{M-Sum-L}\ \frac{\lfloor m_1 + m_2 \rfloor <: m_3}{m_1 + m_2 <: m_3} \qquad \text{M-Sum-R}\ \frac{m_1 <: m_2 \vee m_1 <: m_3}{m_1 <: m_2 + m_3}$$

Fig. 3. Subtyping rules for simple multiplicities (left) and sums (right). *This case actually allows τ to be entirely unrelated in m_1 and m_2.

to describe the type of `css()`: it is an overloaded function (i.e., it can be invoked at more than one type, which we model using intersection types $\tau_1 \& \tau_2$ [18]) where the getter must be invoked on a jQuery object containing *exactly one* element, and where the setter must be invoked on a jQuery object containing *at least one* element. This completely solves the ambiguity problem: a getter cannot be ambiguous if it is invoked on exactly one target.

Additionally, it partially resolves the "falling-off" problem: with the additional machinery of Section 4, we can ensure that `$(aTweet).children().next().next().next()` will have type `jQuery⟨0⟨Element⟩⟩`, which means that attempting to call `css()` will result in a static type error: intuitively, `0⟨Element⟩` is not a subtype of `1⟨'e⟩` for any possible type `'e`.

3.2 Subtyping Multiplicities

To make our notion of multiplicities sufficiently flexible, we need to define a "submultiplicity" relation, analogous to subtyping, that defines when one multiplicity is more general than another. The definition is largely straightforward: most of the primitive constructors should clearly subtype covariantly, e.g., $\mathtt{1}\langle\tau\rangle <: \mathtt{1}\langle\tau'\rangle$ if $\tau <: \tau'$, and $\mathtt{1}\langle\tau\rangle <: \mathtt{01}\langle\tau\rangle$. The one exception is that $\mathtt{0}\langle\tau_1\rangle <: \mathtt{0}\langle\tau_2\rangle$ for any types, since in both cases we have nothing.

Sum multiplicities are trickier to subtype. In particular, unlike the simpler multiplicities, the size of a sum is not immediately obvious; instead both arguments must be normalized, and then the whole sum simplified. The rules for sum multiplicities are shown in Fig. 3. Our typechecker uses M-Sums instead of M-Sum-L if possible, to avoid the loss of precision in normalizing sums.

4 Local Structure

Given multiplicities from the previous section, we now must assign types to the navigation functions (e.g., `next()`) and the jQuery `$` function itself such that they produce the desired multiplicities. One heavyweight approach might be to define a full document schema, as in XDuce [11], and then the types for navigation functions are easily determined from that schema (the selection

function is still non-trivial). But this is impossible for most web content, for two reasons. First and foremost, a page may include arbitrary third-party content, and thus its overall schema would not statically be known. Second, even if all HTML were well-formed, HTML markup is not schematically rich: many tags define generic presentational attributes (e.g., "list item", "paragraph", or "table cell"), rather than semantic information (e.g., "tweet author", or "timestamp").

Clearly, global page structure is too onerous a requirement. But abandoning all structure is equally extreme: certainly, developers expect the page to possess some predictable structure. In this section, we propose a lighter-weight, local alternative to global page schemas, and explain what changes are needed in our type language to incorporate it. We then explain how to use this local structure to give precise types to the jQuery navigation APIs. Finally, we show how to obtain local structure types from the type of the `$()` method itself.

4.1 Defining Structure Locally

Local structure allows developers to define the shape of only those sections of content that they intend to access via jQuery. For instance, our running example would be defined as follows:

```
1 (Tweet : DivElement classes = [tweet] optional classes = [starred]
2   (Time : PElement classes = [time])
3   (Author : PElement classes = [author] optional classes = [starred])
4   (Content : PElement classes = [content short]))
5 (Stream : DivElement classes = [stream]
6   <Tweet>+) // One or more Tweets as defined above
```

This local structure declaration defines five types: `Tweet`, `Time`, `Author`, `Content`, and `Stream`. Moreover, these declarations imply several local invariants between code and element structure:

- Each type implies structural information about an element: for example, a `Tweet` is a `DivElement` that is required to have class `.tweet`.
- Type membership can be decided by the declared classes: for example, at runtime, *every* element in the document with class `.tweet` is in fact a `Tweet`, and *no other elements* are permitted to have class `.tweet`. *Any* class by itself suffices to assign a type to an element: for example, `Content` elements can be identified by either `.content` or `.short`.
- Classes need not be unique: `starred` identifies either `Author`s or `Tweet`s. (This weakens the previous invariant slightly; a single class now may identify a set of types, all of which include that class in their declarations.)
- "Classes" are mandatory: `Content` elements will have both `.content` and `.short` classes. (Combined with the previous invariants, for example, all elements with class `.content` are `Content`s and therefore will also have class `.short`.) "Optional classes" and "ids" may or may not be present on elements at runtime.

The full syntax of local structure declarations is given in Fig. 1. Besides the explicit structure seen above, we support two other declarations. First, for legibility

and reuse, types may be referenced by name from other structures: a `Stream` consists of one or more `Tweet`s, as indicated by reference `<Tweet>` and the repetition operator.[3] Second, the `...` construction (not used in this example) indicates the presence of one or more subtrees of no interest to the program's queries (for instance, a document might contain elements for purely visual purposes, that need never be selected by a query).

Incorporating Local Structure as Types. From the perspective of our type language, `Author` is essentially a refinement of the `PElement` type: in addition to the tag name, we know more precisely what selectors might match it. To record these refinements, we add to our type language a new *CSS refinement type* constructor: `Author = PElement @ div.stream>div.tweet>p.time+p.author`, that includes both the base element type (`PElement`) as well as precise information (the `CSS` selector) indicating where `Author`s can be found relative to other known local structures.

To be useful, we must extend the subtyping relation to handle these `CSS` refinement types. First, note that the refinements may be dropped, so that τ `@` s `<:` τ. For example, `Author <: PElement`. Second, we must decide when one `CSS` refinement is a subtype of another. Lerner [12] showed that set operations on `CSS` selectors are decidable; in particular, it is decidable whether for all documents, the nodes matched by one selector are a subset of the nodes matched by another. Accordingly, τ_1 `@` s `<:` τ_2 `@` t holds whenever $\tau_1 <: \tau_2$ and $s \subseteq t$.

4.2 Using Local Structure

The `CSS` refinement types above exploit only part of the information available from local structure definitions. In particular, they do not capture the structural relations *between* elements. For instance, the Twitter stream definition above also implies that:

- A `Tweet`'s children are each of `Time`, `Author` and `Content`. A `Tweet`'s parent is a `Stream`.
- A `Stream`'s children are all `Tweet`s; a `Stream`'s parent is unknown.
- `Time`s, `Author`s and `Content`s have no children, and have a `Tweet` parent.
- A `Time` has an `Author` as its next sibling, and has no previous sibling.
- An `Author` has a `Time` and a `Content` as its previous and next siblings.
- A `Content` has an `Author` as its previous sibling, and no next sibling.

These are precisely the relationships needed to understand the navigation functions in jQuery. Accordingly, we define four primitive type functions, `@children`, `@parent`, `@next` and `@prev`, whose definitions are pieced together from the local structure: in our example,

- `@children(Tweet)` $=$ `1⟨Time⟩` $+\!\!+$ `1⟨Author⟩` $+\!\!+$ `1⟨Content⟩`, and `@parent(Tweet)` $=$ `1⟨Stream⟩`.

[3] We do not yet support the other regular expression operators $*$ and `?`, though they pose no fundamental difficulty.

- `@children(Stream)` = `1+⟨Tweet⟩` and `@parent(Stream)` = `01⟨Element⟩`.
- `@parent(Time)` = `1⟨Tweet⟩`, `@next(Time)` = `1⟨Author⟩`, `@prev(Time)` = `@children(Time)` = `0⟨Element⟩`, and likewise for `Author` and `Content`.

These functions clearly must be primitives in our system, since they are decidedly not parametric, and inspect the structure of their argument.

We may now enhance our jQuery type with navigation functions:

```
parent :   ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@parent⟨'e⟩⟩⟩,
children : ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@children⟨'e⟩⟩⟩,
next :     ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@next⟨'e⟩⟩⟩,
prev :     ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@prev⟨'e⟩⟩⟩,
```

These types, however, are not quite precise enough: if we have a jQuery object containing a single item, asking for its parent should return at most one item, but the types here ensure that we return many items instead. Worse, we may introduce imprecisions that easily could be avoided. In particular, suppose a developer defines two local structures:

```
(A : DivElement classes = [a]
  (B : DivElement classes = [b]))
(C : DivElement classes = [c]
  (D : DivElement classes = [d]))
```

JQuery supports an `add` function, which concatenates two collections into one for further processing across their combined elements. An appropriate type for this function is

```
add : ∀ n <: 0+⟨Element⟩ . ['jq⟨'m⟩] 'jq⟨'n⟩ -> 'jq⟨'m ++ 'n⟩
```

which neatly expresses the "combination" effect of the method. Using this function and the the rules above, the following code only typechecks under a loose bound (comments beginning with a colon are type assertions):

```
var bAndD = /*:jQuery⟨1+⟨B⟩ ++ 1+⟨D⟩⟩*/$(".b").add($(".d"));
var bdParents = bAndD.parent();
```

The type for `parent()` above requires its receiver to have type `jQuery⟨1+⟨'e⟩⟩`. The typechecker must therefore normalize `1+⟨B⟩ ++ 1+⟨D⟩` to `1+⟨B+D⟩`. Then, `@parent` receives `B+D` as its argument, and returns `1+⟨A+C⟩`. The resulting multiplicity describes one or more `A`s or `C`s, but we might have done better: because `bAndD` is known to include both `B`s and `D`s, the code above returns one or more `A`s *and* one or more `C`s. Therefore, we define `@parent` (and the other primitive navigation type functions) over multiplicities directly, rather than over types:

```
parent :   ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@parent⟨'n⟩⟩
children : ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@children⟨'n⟩⟩
next :     ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@next⟨'n⟩⟩
prev :     ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@prev⟨'n⟩⟩
```

Now `@parent` can be passed the original sum multiplicity without approximation, and its returned multiplicities can be correspondingly more precise; the same holds for the other three functions.

This extra precision is particularly crucial when one of the multiplicities degenerates to zero. Consider the following short program:

```
1 var ts = /*:jQuery⟨1+⟨Time⟩⟩*/ ...;
2 var cs = /*:jQuery⟨1+⟨Content⟩⟩*/ ...;
3 var tsAndCs = /*:jQuery⟨1+⟨Time⟩⟩+1+⟨Content⟩⟩*/ ts.add(cs);
4 var tsOrCs = /*:jQuery⟨1+⟨Time+Content⟩⟩ */ tsAndCs; // looser type
5 var prevTsAndCs = tsAndCs.prev(); // Can be jQuery⟨1+⟨Author⟩⟩
6 var prevTsOrCs  = tsOrCs.prev();  // Must be jQuery⟨0+⟨Author⟩⟩
```

The combined collection `ts.add(cs)` can be given two types: one using sum-multiplicities and one using a normalized multiplicity with a union type. When calling `@prev(1+⟨Time⟩⟩+1+⟨Content⟩)`, `@prev` can distribute over the sum and since we know that there exists at least one `Content` element, the result must contain at least one `Author`. But when calling `@prev(1+⟨Time+Content⟩)`, we might have a collection containing only `Time` elements and so the result collection may be empty.

4.3 Creating Local Structure with the `@selector` Function

The last remaining challenge is to ascribe a type to the `$` function itself. We start with two concrete examples, and then explain the general case. Finally, we explore some subtleties of our design.

CSS and the `@selector` Function. Consider determining which local structures matches the query string `"* > .author"`. We examine the local structure for types that mention the class `.author`, finding just one in this case, namely `Author = PElement @ div.stream > div.tweet > p.time + p.author`. We must then check whether the offered selector `* > .author` and the structure's selector `div.stream > div.tweet > p.time + p.author` can ever describe the same element: this is an intersection test over selectors, and can be done easily [12]. In fact, `* > .author` does intersect with `Author`'s selector (because all `Author`s do have a parent element). Finally, recall that the local structure invariants in Section 4.2 implicitly assert that, by omission, all other local structures are asserted *not* to have class `.author`, and are therefore excluded. We can therefore conclude `@selector⟨"div > .author"⟩ = 1+⟨Author⟩`.

On the other hand, consider the selector `"div.author"`. We know that `"div.author"` does not intersect with `Author` (because an element cannot match both `"div"` and `"p"`), nor with anything else (because nothing else has class `".author"`), so `@selector⟨"div.author"⟩ = 0⟨Element⟩`.

As with the primitive navigation type functions, we can encapsulate this reasoning in another primitive function, `@selector`. Here we must pass the precise string value of the selector to the type function, so that it can be examined

as needed. We exploit Typed JS's refined string types: rather than grouping all strings together under type String, Typed JS uses regular expressions to define subtypes of strings [9]. (In fact, String is just an alias for the regular expression /.*/, which matches all strings.) In particular, string literals can be given singleton regular expression types matching only that string, and so pass the string value into our type function.[4] Therefore, we might give the $ function the general type

```
1 $ : ∀ s <: /.*/ . 's -> jQuery⟨@selector⟨'s⟩⟩
```

The full algorithm for @selector, then, parses the string argument as a CSS selector. When it fails, it simply returns 0⟨Element⟩. It next searches through the local structure definitions for candidates with matching class names, and intersects the query selector with the selectors compiled from the candidates' local structure declarations (using the approach in [12]). The returned *type* of @selector is the union of those local structure types whose selectors intersect the query string. (But see Section 5 below for which *multiplicity* it should return.)

Note that we never expose the parsing and interpretation of strings as CSS queries directly, but only the result of comparing strings to the selectors induced by local structure. We also do not add type constructors to our type language that mimic CSS selector combinators; we instead use only the refinement types shown above. We chose this design to keep our type system loosely coupled from the precise query language being used; our only requirement is that query intersections and containment be decidable. If jQuery used another language, perhaps more expressive than CSS, we would only need to replace the CSS selector intersection and containment algorithms, and not need to change any other aspects of our type system.

Applying @selector to Overly-Broad Queries. Defining the @selector function entails two crucial choices: how many items should it return when the query matches local structures, and how flexible should it be in matching selectors to local structure? The former question is quite subtle, and we address it in Section 5. The latter is more a matter of programmer expectations, and we can resolve it relatively easily.

In our Twitter stream example, what should the query $("div > p") return? It does not mention any local structure classes or ids, so we have three options:

1. We might return 0+⟨PElement @ div > p⟩, because nothing in the selector matches any of the required classes or ids of the local structure.
2. We might return 1+⟨Time + Author + Content⟩, because each of these three structure definitions match the query selector, even though none of the required classes are present in the query.

[4] Note: we do not *require* the argument to the $() function to be a string literal; the types here admit any string expression. We attempt to parse that type as a CSS selector, and such precise types often only work to literals. Since in practice, the arguments we have seen usually *are* literals, this encoding most often succeeds.

3. We might return the "compromise" union `0+⟨Time + Author + Content + (PElement @ div>p)⟩`, because the query might return either any of the declared structures, or any other ⟨**p**/⟩ elements with ⟨**div**/⟩s as parents.

Of the three options, the third is clearly the least appealing, as it is tantalizingly useful (including `Time`, `Author` and `Content`), while still not guaranteeing anything about the returned values' multiplicity or type. The second is the most precise and useful, but it is incorrect: there might well be other elements matching `div > p` that are not part of a `Tweet` structure. As a result, the only sound multiplicity we can return in this case is the first one, which has the side benefit of highlighting, by its obtuseness, the imprecision of the query selector.

Unfortunately, several of the examples we evaluated used queries of this vague and overly-broad form! An amateur jQuery programmer, upon reading these examples, might infer that query brevity is preferred over precision. To accommodate these examples, we define our selector function to return the first—sound—option above by default, but currently we provide a command-line flag to implement the second (unsound but useful) option instead. (But see Section 5 below for a non-flag-based solution.) We view the need for this option as an unfortunate symptom of web developers' current practice of writing jQuery code without co-developing a specification for that code.

5 Type-System Guarantees and Usability Trade-Offs

JQuery provides four APIs for querying: `$` itself, that selects nodes in the document; `find()`, that takes a current query set and selects descendant nodes; `filter()`, that selects a subset of the current query set; and `eq()`, that selects the n^{th} element of the current query set. Each of these might return zero items at runtime, so the most natural types for them are:

```
1 $ : ∀ s <: String . 's -> jQuery⟨0+⟨@selector⟨'s⟩⟩⟩
2 filter : (∀ e <: Element, ['jq⟨0+⟨'e⟩⟩] ('e -> Bool) -> 'jq⟨0+⟨'e⟩⟩) &
3          (∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨'m, 's⟩⟩⟩)
4 find : ∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨@desc⟨'m⟩, 's⟩⟩⟩
5 eq : ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] Num -> 'jq⟨01⟨'e⟩⟩
```

(In these types, `@filter` is a variant of `@selector`, and `@desc` is the transitive closure of `@children`.) But these types are inconvenient: because they all include `0`, their return values cannot be chained with methods that require a non-zero multiplicity. Note that these zeroes occur *even if the query string matches local structure*. This is particularly galling for the `$` function, as after all, if a query string *can* match local structure definitions, then surely the developer can expect that it *will* actually select nodes at runtime! Worse, using this type effectively marks all jQuery chains as type errors, because the first API call after `$()` will expect a non-zero multiplicity as input. Likewise, developers may often expect that their `filter()` or `find()` calls will in fact find non-zero numbers of nodes.

Despite their precision, our types seem to have lost touch with developers' expectations. It appears the only way to make our types useful again is to change

them so they no longer reflect the behavior of jQuery! What guarantee, then, does our type system provide, and how can we resolve this disparity between "expected" and actual jQuery behavior?

5.1 Type-System Guarantees

We have described our type system above as capturing jQuery's behavior "precisely", but have not formalized that claim. Recall that the traditional formal property of *soundness* asserts that a type system cannot "lie" and affirm that an expression has some type when in fact it does not, or equivalently, affirm that an expression will execute without error at runtime when in fact it will. Such soundness claims are necessarily made relative to the semantics of the underlying language. But jQuery is a language without a formal semantics, so what claims besides soundness can we make about our approach?

Our type system is based upon Typed JS [10], which has a formal soundness guarantee relative to JS semantics. We are therefore confident that our system correctly handles typical JS features. However, while our type system may be sound for JS, we might ascribe types to jQuery APIs that do not match their actual behavior. This would be a failing of our *type environment*, not of our type system. To produce plausibly correct types for jQuery's APIs, we have experimented with many examples to determine potential types, and constructed counterexamples to convince ourselves the types cause typing errors only in programs we expected to be wrong.

Of course, jQuery is in fact implemented in JS, and therefore it is conceivable that we might typecheck the implementation to confirm it has the types we have ascribed. As our experience with typechecking ADsafe [16] has shown, typechecking a library can be a complex undertaking in its own right; we leave typechecking the source of jQuery itself as future work we intend to pursue.

But as we have begun to see already in Footnote 4, even choosing appropriate types for jQuery involves aesthetic choices, trading off between strictness and static determination of errors on one hand, and flexibility and ease of development on the other. We now examine these trade-offs in more detail.

5.2 Accommodating Varied Developer Expectations

The useful output of a type system—its acceptance or rejection of a program, and the error messages that result—provides crucial feedback to developers to distinguish buggy code from incomplete code. But type checkers can only operate on the code that is actually written, and that may not always suffice to distinguish these two cases, particularly when they are syntactically identical.

Merely by writing queries, developers have codified implicit expectations about how the state of their program matches their code. In particular, they understand that because of the dynamic, stateful changes to page structure that make their application interactive, the size of a query's result may vary during execution: queries that might return several elements at one point of a program's execution might later return none, or vice versa. But buggy queries

also may have unexpected sizes. Nothing syntactically distinguishes queries with anticipated size variability from ones that are buggy. Without this knowledge, we cannot reliably report type errors to developers. We explore two untenable approaches for resolving this ambiguity, and then explore two more palatable ones.

A Non-solution: Specialized Code. One unsuccessful strawman approach might require programmers to write repetitive variants of code to deal with each potential query result size. This is untenable, as these variants clutter the code, make future code modifications difficult, and break from the compositional style of jQuery programs.

An Unsound "Solution": Assuming Presence. Another approach might simply assert by fiat that any local structures defined by the author are always guaranteed to be present: effectively, this means removing the explicit `0+` in the output type of the query APIs. Under this assumption, the following query should always return `1⟨Tweet⟩` (recall the definition of `Tweet`s from Section 4):

```
1 $(".tweet").find(".starred").first() // Has type jQuery⟨1⟨Tweet⟩⟩
```

But the developer explicitly said `starred` only *might* be present! Further, it is laughably wrong in the face of page mutations:

```
1 $(".tweet").find(".starred").removeClass("starred");
2 $(".tweet").find(".starred").first() // Still has type jQuery⟨1⟨Tweet⟩⟩
```

Note that this assumption leads to incorrect types, in that it does not accurately reflect the behavior of `find()`. But this type does not break the soundness of the type system itself: it is entirely possible to write a method that always returns a non-zero number of elements—though to do so, one would have to use raw DOM APIs, and effectively make such a function a "primitive" operation with respect to the jQuery language.

On the other hand, these "incorrect" types are much more useful for the `$()` function: by allowing non-zero multiplicities, jQuery chains can be checked without immediately requiring annotation.

A Spectrum of Type Environments. One seemingly simple possibility is simply to add "flags" to the type system indicating whether `find()` (above), `@selector` (as in Footnote 4), and many other functions should be strict or permissive about their multiplicities. We reject this approach. If the flags affect the core type language or the type checker itself, then any soundness or correctness claims of the type system can only be made relative to every possible combination of flags. As type checkers are already quite complicated, these flags are a poor engineering decision, especially when a much cleaner, modular alternative exists.

Rather than add these flags to the type *system*, we can compile a variety of *type environments* from local structure, that contain stricter or looser types

for these functions. Unlike "flags", whose interpretation is hidden within the type system's implementation, these various environments are easily examined by the developer, who can see exactly how they differ. A developer choosing to use the looser environments is therefore making a conscious choice that certain error conditions are not important, or equivalently that he is willing for them to appear as runtime errors instead. Moreover, the developer may migrate from permissive to strict environments as his program matures and nears completion.

The Need for New APIs. Our last approach requires actually enhancing the jQuery API. JQuery objects expose their size via the `length` field; we might use this to regain lost precision. We might force programmers to explicitly write if-tests on the `length` field, but this has the unwanted effect of breaking chaining. Instead, we propose new jQuery APIs:

```
assertNotEmpty : (∀ t, ['jq⟨01⟨'t⟩⟩] -> 'jq⟨1⟨'t⟩⟩) &
                 (∀ t, ['jq⟨0+⟨'t⟩⟩] -> 'jq⟨1+⟨'t⟩⟩)
ifZero : ∀ t, ['jq⟨0+⟨'t⟩⟩] (['jq⟨0⟨'t⟩⟩]->Undef) -> 'jq⟨0+⟨'t⟩⟩
ifOne  : ∀ t, ['jq⟨0+⟨'t⟩⟩] (['jq⟨1⟨'t⟩⟩]->Undef) -> 'jq⟨0+⟨'t⟩⟩
ifMany : ∀ t, ['jq⟨0+⟨'t⟩⟩] (['jq⟨1+⟨'t⟩⟩]->Undef) -> 'jq⟨0+⟨'t⟩⟩
```

The first API converts possibly-empty collections into definitely-non-empty ones, and throws a runtime error if the collection is in fact empty. Developers would use this API to indicate queries they expect should never fail. The example above would be rewritten

```
$(".tweet").find(".starred").removeClass("starred");
$(".tweet").find(".starred").assertNotEmpty().first();
```

This would typecheck successfully, and always throw an exception at runtime. By contrast, the latter three APIs take a receiver object of unknown multiplicity, and a callback that expects a argument of precise multiplicity, and calls it only if the receiver has the expected multiplicity. Developers would use these APIs to indicate computation they expect might not be needed. The example above would be rewritten

```
$(".tweet").find(".starred").removeClass("starred");
$(".tweet").find(".starred").ifMany(function() { this.first(); });
```

This again would typecheck successfully, and would not call the inner function at runtime. Unlike the previous API, these APIs never throw exceptions, but indicate to the type checker that the programmer knows that these calls—and these alone—might fail. These APIs implicitly perform a specialized form of occurrence typing [22], without needing any extra type machinery.

6 Modeling JQuery Reality

Real jQuery programs avail themselves of several additional query APIs. One of them requires revising the kind of our type, and leads to the final form of our

type for jQuery. This change unfortunately makes the type more unwieldy to use, but we can resolve this with a simple type definition.

Merging and Branching. We have already seen the `add()` operation, which allows combining two jQuery objects into one. Occasionally, a program might need the dual of this operation, "splitting" one jQuery into multiple subqueries. Consider the following two queries:

```
$(".tweet").find(".star").each(processStarredTweet);
$(".tweet").find(".self").each(processOwnTweet);
```

They both have the same prefix, namely `$(".tweet")`, and more generally this prefix might be an expensive computation. To avoid this, jQuery objects conceptually form a stack of their navigated states, where each navigation pushes a new state on the stack: programs may use a new API, `end()`, to pop states off this stack. The example above would be rewritten:

```
$(".tweet").find(".star").each(processStarredTweet)
           .end() // pops the find operation
           .find(".self").each(processOwnTweet);
```

Internally, jQuery objects form a linked list that implements this stack: the chainable jQuery methods that appear to modify the contents of the collection return a new jQuery object that wraps the new contents and whose tail is the old jQuery object. (The old object is sill available, and has not been mutated.) The `end()` method simply returns the tail of the list. In order to type `end()` correctly, we must encode the linked list in our types. To do so requires a systematic change in our type: we redefine the `jQuery` type constructor to take *two* type parameters, where the new second type parameter describes the tail of the stack. Our final type definition for jQuery is (only representative examples of jQuery's more than fifty query APIs are shown; the full type is available in our implementation):

```
type jQuery = μ jq :: (M⟨∗⟩, ∗) => ∗ .
  Λ m :: M⟨∗⟩, prev :: ∗  . {
    // Navigation APIs: next, prev, parent, etc. are analogous
    children : ∀ e <: 1+⟨Element⟩ .
          ['jq⟨'e, 'prev⟩] -> 'jq⟨@childrenOf⟨'e⟩, 'jq⟨'e, 'prev⟩⟩
    // Accessor APIs: offsetX, height, attr, etc. are analogous
    css : ∀ e <: Element .
          (['jq⟨1⟨'e⟩, 'prev⟩] Str -> Str &
           ['jq⟨1+⟨'e⟩, 'prev⟩] Str∗Str -> 'jq⟨1+⟨'e⟩, 'prev⟩)
    // Stack-manipulating APIs
    add : ∀ n <: 0+⟨Element⟩, 'q .
          ['jq⟨'m, 'prev⟩] 'jq⟨'n, 'q⟩ -> 'jq⟨'m++'n, 'jq⟨'m, 'prev⟩⟩
    end : ['jq⟨'m, 'prev⟩] -> 'prev
    // Collection-manipulating APIs
    each : ∀ e <: Element . ['jq⟨0+⟨'e⟩, 'prev⟩] ('e->Undef) -> 'jq⟨'m, 'prev⟩
    filter : (∀ e <: Element, ['jq⟨0+⟨'e⟩⟩] ('e->Bool) -> 'jq⟨0+⟨'e⟩⟩) &
             (∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨'m, 's⟩⟩⟩)
    find : ∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨@desc⟨'m⟩, 's⟩⟩⟩
    eq : ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] Num -> 'jq⟨01⟨'e⟩⟩
    // No other fields are present
    ⋆ : _
  }
```

Methods such as `add()` and `children()` nest the current type one level deeper, and `end()` simply unwraps the outermost type.

Convenience. Often the suffix of the stack is not needed, so we can define a simple synonym for the supertype of all possible jQuery values:

```
type AnyJQ = ∀ p . jQuery⟨0+⟨Any⟩, 'p⟩
```

Any functions that operate over jQuery objects, and that will not use `end()` to pop off the jQuery stack any more than they push on themselves, can use `AnyJQ` to summarize the type stack easily.

7 Evaluation

To determine the flexibility of our type system, we analyzed a dozen samples from *Learning JQuery* [6] and from the Learning JQuery blog [20, 21]. Our prototype system[5] supports standard `CSS` selectors, but not pseudoselectors or the bespoke jQuery extensions to `CSS` syntax. In the examples below, we have edited the original code by replacing pseudoselectors and extensions with calls to their API equivalents and with calls to `ifMany()` (defined above). These added calls represent our understanding of how the examples should work; without them, the program's intended behavior is unclear. With these added calls, the examples below were checked *assuming presence for the `$()` function, but not for `find()` or `filter()`*. We distinguish three sets of results: those that should typecheck and do, those that should not typecheck and do not, and cases that expose weaknesses for our system.

7.1 Successfully Typechecking Type-Correct Examples

Our samples consist of between 2 and 9 calls to jQuery functions in linear, branching, and nested call-patterns. These examples *each typecheck with no annotations needed on the code* besides local structure definitions that need only be written once. These local structures were *not defined* by the accompanying text; we derived them by manual inspection of the examples and their intended effects. We highlight three illustrative examples here.

Selecting All of a Row. Our first example shows a straightforward selection of all ⟨**td**/⟩ tags in a table row and adding a class to them. Note that `addClass()` is called with a class name that is not declared in the local structure; our type system is not intended to restrict such code. Rather, it ensures that the call to `addClass()` is given a non-empty collection to work with. Note also that the initial query is over-broad (a better query would be `'td.title'`) and requires the command-line option for flexible query matching (of Footnote 4) to typecheck as written:

[5] Available at `http://www.jswebtools.org/jQuery/`

```
/*::
  (PlaysTable : TableElement ids = [playstable] classes = [plays]
    (PlaysRow : TRElement classes = [playsrow]
      (Title : TDElement classes = [title])
      (Genre : TDElement classes = [genre])
      (Year : TDElement classes = [year])
    )+ );
*/
$('td')      // Has type t1 = jQuery⟨1+⟨Title⟩++1+⟨Genre⟩++1+⟨Year⟩, AnyJQ⟩
  .filter(':contains("henry")')   // t2 = jQuery⟨0+⟨Title+Genre+Year⟩, t1⟩
  .ifMany(function() {      // [jQuery⟨1+⟨Title+Genre+Year⟩, t1⟩] -> Undef
    this                          // t3 = jQuery⟨1+⟨Title+Genre+Year⟩, t1⟩
      .nextAll()                 // t4 = jQuery⟨1+⟨Genre⟩ ++ 1+⟨Year⟩, t3⟩
      .andSelf()          // jQuery⟨1+⟨Genre⟩ ++ 1+⟨Year⟩ ++ 1+⟨Title⟩, t4⟩
      .addClass('highlight');            // allowed, non-empty collection
   });
```

Filtering by Ids. This next example is a deliberately circuitous query, and illustrates several features both of jQuery and our type system. The call to `parents()` normally would return all the known parents of the current elements, up to and including the generic type `Element`, but because the query also filters on an id, and because that id appears in local structure, our type is much more tightly constrained. Similarly, the calls to `children()` and `find()` are constrained by the local structure. Note that if the call to `parent()` were *not* so constrained, then these subsequent calls would mostly be useless, since nothing is known about arbitrary `Element`s.

```
/*::
(SampleDiv : DivElement ids = [jqdt2] classes = [samplediv]
  (Paragraph : PElement classes = [goofy]
    ...)              // The children of a Paragraph are irrelevant here
  (OrderedList : OLElement classes = [list]
    (LinkItem : LIElement classes = [linkitem]
      (Link : AElement classes = [link]))
    (GoofyItem : LIElement classes = [goofy]
      (StrongText : StrongElement classes = [strongtext]))
    (FunnyItem : LIElement classes = [funny])
    <LinkItem>
    <GoofyItem>));
*/
$('li.goofy')               // Has type t1 = jQuery⟨1+⟨GoofyItem⟩, AnyJQ⟩
  .parents('#jqdt2')                    // t2 = jQuery⟨1+⟨SampleDiv⟩, t1⟩
  .children('p')                        // t3 = jQuery⟨1+⟨Paragraph⟩, t2⟩
  .next()                              // t4 = jQuery⟨1+⟨OrderedList⟩, t3⟩
  .find('a')                                 // t5 = jQuery⟨1+⟨Link⟩, t4⟩
  .parent();                               // t6 = jQuery⟨1+⟨LinkItem⟩, t5⟩
```

Manipulating Each Element. The following example demonstrates that our approach scales to higher-order functions: the callback passed to the `each()`

operation (line 16) needs no annotation to be typechecked. The `ifMany()` calls could be eliminated using the unsound options described in Section 5. Note also the use of `end()` on line 14 to restore a prior query state.

```
1  /*::
2  (NewsTable : TableElement classes = [news] ids = [news]
3    ...                                  // Placeholder element; never queried
4    (NewsBody : TBodyElement classes = [newsbody]
5      (YearRow : TDElement classes = [yearrow])
6      (NewsRow : TRElement classes = [newsrow] optional classes = [alt]
7        (NewsInfo : TDElement classes = [info])+)+
8      <YearRow>
9      <NewsRow>+))
10 */
11 $('#news')                      // Has type t1 = jQuery⟨1⟨NewsTable⟩, AnyJQ⟩
12   .find('tr.alt')                              // t2 = jQuery⟨1+⟨NewsRow⟩, t1⟩
13   .ifMany(function() { this.removeClass('alt'); })                  // t2
14   .end()                                                            // t1
15   .find('tbody')                               // t3 = jQuery⟨1⟨NewsBody⟩, t1⟩
16   .each(function() {                                  // [NewsBody] -> Undef
17      $(this)                                 // t4 = jQuery⟨1⟨NewsBody⟩, AnyJQ⟩
18        .children()             // t5 = jQuery⟨1+⟨YearRow⟩ ++ 1+⟨NewsRow⟩, t4⟩
19        .filter(':visible')      // t6 = jQuery⟨0+⟨YearRow⟩++0+⟨NewsRow⟩, t5⟩
20        .has('td')                              // t7 = jQuery⟨0+⟨NewsRow⟩, t6⟩
21        .ifMany(function() {           // [jQuery⟨1+⟨NewsRow⟩, t6⟩] -> Undef
22          this.addClass('alt');             // allowed, non-empty collection
23        });                                                           // t7
24   });                                                                // t3
```

7.2 Successfully Flagging Type-Incorrect Examples

The examples we examined from the *Learning JQuery* textbook and blog are typeable. To verify that our type-checker was not trivially passing all programs, we injected errors into these examples to ensure our system would correctly catch them, and it does. Our modifications changed queries to use the wrong element id, or have too many or too few navigational calls.

```
1  $('li.goofy')                // Has type t1 = jQuery⟨1+⟨GoofyItem⟩, AnyJQ⟩
2    .parents('#WRONG_ID')       // t2 = jQuery⟨01⟨Element @ "#WRONG_ID"⟩, t1⟩
3    .children('p');
4          ⇒ ERROR: children expects 1+⟨Element⟩, got 01⟨Element⟩
5  $('#news')                     // Has type t3 = jQuery⟨1⟨NewsTable⟩, AnyJQ⟩
6    .find('tr.alt')                              // t4 = jQuery⟨1+⟨NewsRow⟩, t3⟩
7    .ifMany(function() { this.removeClass('alt'); })                  // t4
8                                          // Note: missing call to .end()
9    .find('tbody')                               // t5 = jQuery⟨0⟨Element⟩, t3⟩
10   .each(...)
11         ⇒ ERROR: each expects 1+⟨Element⟩, got 0⟨Element⟩
12 $(".tweet").children().next().next().next().css("color", "red");
```

```
13          ⇒ ERROR: css expects 1+⟨Element⟩, got 0⟨Element⟩
14 $(".tweet").children().next().css("color");
15          ⇒ ERROR: css expects 1⟨Element⟩, got 1+⟨Author + Time⟩
```

7.3 Weaknesses of Our System

Beyond the restrictions shown in the paper, and the trade-offs between soundness and utility of types, there still exist queries our system cannot handle. We construct one such instance here.

Under our running example, the expression `$(".tweet").parent()` will have the expected type `jQuery⟨1+⟨Stream⟩, AnyJQ⟩`. However, the similar expression `$(".stream").parent()` correctly has type `jQuery⟨0+⟨Element⟩, AnyJQ⟩`, because nothing is known about the parent of a `Stream`.

The expression `$("div.outer > *.stream").parent()` will at runtime return the ⟨**div** div="outer"/⟩ elements that contain streams (if any exist). Our current type system, however, will nevertheless give this expression the type `jQuery⟨0+⟨Element⟩, AnyJQ⟩`, even though *these* `Stream`s definitely have ⟨**div**/⟩s as parents. This is inherent in our semantics for local structure: developers are obligated to provide definitions for any content they intend to access via jQuery's APIs—beyond those definitions, the remainder of the page is unknown. One might re-engineer our system to dynamically refine the local structure-derived types to include the extra information above their root elements, but such complication both muddies the implementation and also confuses expectations of what the system can provide developers. Instead, developers must simply provide extra local structure information, and avoid the confusion entirely.

8 Related Work

XDuce and ℂDuce. Languages such as XDuce [11, 19] and ℂDuce [2] embed an XML-processing language into a statically-typed general-purpose language, and extend the type language with document schemas. These languages differ from our jQuery setting in three crucial ways, all stemming from the fact that jQuery is a language for manipulating HTML that is embedded within JS.

First and foremost, XDuce and ℂDuce operate over well-schematized databases represented in XML, from which richly-typed structured data can be extracted. But HTML is not schematized: it is almost entirely free-form, and largely presentational in nature. As we argued in Section 4, mandating a global schema for HTML documents is an untenable requirement on developers. As such, XDuce and ℂDuce's approach cannot apply directly.

Second, XDuce and ℂDuce go to great lengths to support pattern matching over XML: in particular, their language of values includes *sequences* of tagged values, i.e. XML forests. XDuce and ℂDuce define *regular expression types* to precisely type these sequences. But jQuery does not have the luxury of enriching JS to support free-form sequences: instead, it encapsulates a sequence of values into a JS object. As the ℂDuce paper notes [2], regular expression types are not themselves

types, but are only meaningful within the context of an element: this is exactly comparable to our kinding distinction between multiplicities and types.

Third, the operations native the HTML and DOM programming are simpler than those in XML processing: web programmers use CSS selectors [25] to query nodes in their documents, rather than XPath. Further, because JS has no pattern-matching construct, there is no need for an analysis of jQuery to define tree patterns or regular tree types (as in [11]), or check exhaustiveness of pattern matching (as in [5]). Instead, as we have shown, a simpler notion suffices.

Semanticizing the Web. This work fits into a growing effort to design—or retrofit—semantics onto portions of the client-side environment, including JS [9, 14, 15], the event model [13], and the browser itself [3]. Some work has begun addressing the semantics of DOM manipulations [8], but it has focused on the low-level APIs and not on the higher-level abstractions that developers have adopted.

Structure and Size Types. Other work [26] extends Haskell's type system with indexed types that describe sizes statically. Multiplicities specialize this model by admitting ranges in the indices. Our implementation also demonstrates how to use multiplicities without annotation burden, in the context of an important web library.

9 Future Work: Interoperating with Non-jQuery Code

While our type system cleanly and accurately supports the query and navigation portions of jQuery, other challenges remain in analyzing the interactions between jQuery and raw JS code. We focus on two interactions: code written with types but without multiplicities, and code that might breach local structure invariants.

Relating Types and Multiplicities. Multiplicities, as presented so far, are strictly segregated from types by the kinding system. However, suppose we had two nearly-identical list types, defined by client code and by a library:

```
1 type AMultList = Λ m :: M⟨∗⟩ . ...
2 type ATypeList = Λ t :: ∗ . ...
```

A value of type `AMultList`$\langle$`1+`$\langle\tau\rangle\rangle$ cannot be used where one of type `ATypeList`$\langle\tau\rangle$ is expected, and yet any program expecting the latter would behave correctly when given the former: by its type, it clearly cannot distinguish between them. However, upon returning from non-multiplicity-based code, we have to construct some multiplicity from a given type. Accordingly, we might add the two "subtype/multiplicity" rules

$$\textsc{Lax-TypMult}\ \frac{}{\tau <: \texttt{0+}\langle\tau\rangle} \qquad \textsc{Lax-MultTyp}\ \frac{}{\texttt{0+}\langle\tau\rangle <: \tau}$$

These two rules let us be "lax" about keeping multiplicities and types separate, without sacrificing soundness. Note that the utility of Lax-TypMult depends heavily on using occurrence typing to refine the resulting $\texttt{0+}\langle\tau\rangle$ multiplicity into something more precise. We have implemented these rules in our type checker, but they have not been needed in the examples we have seen.

Preserving Local Structure. Our type for `jQuery` ensures only that developer-supplied callbacks typecheck with the appropriate receiver and argument types (see Section 3). This is an incomplete specification, as such excursions into low-level JS can easily be sources of bugs. We envision both dynamic and static solutions to this problem.

Dynamically, maintaining the local structure of a subset of a page amounts to a contract. The local structure definitions can easily be compiled into executable JS code that checks whether a given node conforms to particular local structure definitions, and these checks can be automatically wrapped around all jQuery APIs.

Statically, maintaining tree shapes may be analyzable by a tree logic [7]. The primary challenge is a reachability analysis identifying all the possible nodes that might be modified by arbitrary JS; we leave this entirely to future work.

References

1. Abel, A.: Polarized subtyping for sized types. Mathematical Structures in Computer Science 18(5), 797–822 (2008)
2. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. In: ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 51–63. ACM, New York (2003)
3. Bohannon, A., Pierce, B.C.: Featherweight Firefox: formalizing the core of a web browser. In: USENIX Conference on Web Application Development (WebApps), pp. 123–134 USENIX Association, Berkeley (2010)
4. BuiltWith. JQuery usage statistics, `http://trends.builtwith.com/javascript/JQuery` (retrieved November 2012)
5. Castagna, G., Colazzo, D., Frisch, A.: Error mining for regular expression patterns. In: Coppo, M., Lodi, E., Pinna, G.M. (eds.) ICTCS 2005. LNCS, vol. 3701, pp. 160–172. Springer, Heidelberg (2005)
6. Chaffer, J., Swedberg, K.: Learning JQuery, 3rd edn. Packt Publishing Ltd., Birmingham (2011)
7. Gardner, P., Wheelhouse, M.: Small specifications for tree update. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 178–195. Springer, Heidelberg (2010)
8. Gardner, P.A., Smith, G.D., Wheelhouse, M.J., Zarfaty, U.D.: Local Hoare reasoning about DOM. In: ACM SIGMOD Symposium on Principles of Database Systems (PODS), pp. 261–270. ACM Press, New York (2008)
9. 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)
10. Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing local control and state using flow analysis. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 256–275. Springer, Heidelberg (2011)

11. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology (TOIT) 3(2), 117–148 (2003)
12. Lerner, B.S.: Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design. Ph.D. thesis, University of Washington Computer Science & Engineering (August 2011)
13. Lerner, B.S., Carroll, M.J., Kimmel, D.P., de la Vallee, H.Q., Krishnamurthi, S.: Modeling and reasoning about DOM events. In: USENIX Conference on Web Application Development (WebApps). USENIX Association, Berkeley (2012)
14. 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)
15. Politz, J.G., Carroll, M., Lerner, B.S., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in JavaScript. In: Dynamic Languages Symposium, (DLS) (2012)
16. Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: ADsafety: type-based verification of JavaScript sandboxing. In: USENIX Security Symposium, p. 12. USENIX Association, Berkeley (2011)
17. Politz, J.G., Guha, A., Krishnamurthi, S.: Semantics and types for objects with first-class member names. In: Workshop on Foundations of Object-Oriented Languages, (FOOL) (2012)
18. St-Amour, V., Tobin-Hochstadt, S., Flatt, M., Felleisen, M.: Typing the numeric tower. In: Russo, C., Zhou, N.-F. (eds.) PADL 2012. LNCS, vol. 7149, pp. 289–303. Springer, Heidelberg (2012)
19. Sulzmann, M., Lu, K.Z.M.: A type-safe embedding of XDuce into ML. In: Workshop on ML, pp. 229–253. ACM Press, New York (2005)
20. Swedberg, K.: How to get anything you want - part 1, `http://www.learningjquery.com/2006/11/how-to-get-anything-you-want-part-1` (written November 2006)
21. Swedberg, K.: How to get anything you want - part 2, `http://www.learningjquery.com/2006/12/how-to-get-anything-you-want-part-2` (written December 2006)
22. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 395–406. ACM Press, New York (2008)
23. W3C. XML path language (XPath) 2.0, `http://www.w3.org/TR/xpath20/` (written December 2010)
24. W3C. XQuery 1.0: An XML query language, `http://www.w3.org/TR/xquery/` (written December 2010)
25. W3C. Selectors level 3, `http://www.w3.org/TR/selectors/` (written September 2010)
26. Zenger, C.: Indexed types. Theoretical Computer Science 187(1-2), 147–165 (1997)

Heap Decomposition Inference with Linear Programming

Haitao Steve Zhu and Yu David Liu

SUNY Binghamton
Binghamton NY 13902, USA
{hzhu1,davidL}@cs.binghamton.edu

Abstract. Hierarchical decomposition is a fundamental principle that encourages the organization of program elements into nested scopes of access, instead of treating all as "global." This paper offers a foundational study of *heap decomposition inference*, the problem of statically extracting the decomposition hierarchy latent in the runtimes of object-oriented programs, henceforth revealing the compositional nature of the heap. The centerpiece of the paper is Cypress, a sound, precise, and scalable constraint-based ownership type inference coupled with a novel application of linear programming over integers. All constraints in Cypress are linear, and the precision of decomposition – placing objects to scopes as non-global as possible – can be reduced to a linear programming problem. Cypress has been implemented as an open-source tool that can decompose real-world Java applications of more than 100K LOC and up to 6000 statically distinct instantiations.

1 Introduction

The principle of hierarchical decomposition (HD) is a divide-and-conquer strategy to compartmentalize program element accesses into potentially nested scopes. Language support for HD over static program elements – such as lexical scoping – has a history as long as programming language design itself. HD over program runtimes are a direction particularly successful through language design such as ownership type systems [29,6,5,2,3,11].

This paper explores a program analysis approach to HD, addressing how the heap of the standard Java-like object model can be decomposed into a hierarchy that reflects the principle of HD. In other words, even though the heap of Java-like languages is a *global* structure where every object may potentially be referred to by all others, few practical programs take advantage of that potential. Instead, it is more natural to consider every heap object to be implicitly associated with a scope – the same concept as in HD – and view the heap as a *compositional* tree structure where each object serves as the scope of access for its children on the tree. The goal of the inference is to compute a static approximation of the transformation from a global heap to a compositional heap, informing programmers of what their heap "could have been."

Our concrete proposal is Cypress, a sound, precise, and scalable constraint-based type inference. As a form of ownership inference [14,28,34,26,10,17], Cypress is distinct with the following features:

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 104–128, 2013.

- *Linear constraints for scalability*. Cypress constraints for computing the compositional heap are *linear*, a scalable solution outsourceable to mature linear solvers.
- *Linear programming for precision*. The precision of decomposition – placing objects to scopes as non-global as possible – is guided by a novel *Cypress Principle* [1], which says a "tall and skinny" view of the compositional heap is favored, and this preference can be reduced to a linear programming problem with a judicious setting of the objective function.
- *Heap-wide scope characterization*. Cypress computes scopes for all heap objects in an algorithm that does not require iteration over each object. The result of the analysis is a tree including all (static) heap objects, organized in a way vividly reflecting how the heap is decomposed according to the principle of HD.
- *Fully automated static inference*: Cypress is a sound type inference that does not require programmer annotation or interactive user assistance.

Beyond the technical contribution of proposing a new ownership inference algorithm, the paper is further aimed at gaining a fundamental understanding of HD as a principle. It provides a unified framework where commonly discussed HD mechanisms – lexical decomposition (lexical scoping), dynamic heap decomposition (dynamic ownership), and static heap decomposition (static ownership) – are formally related, and the HD principle is captured as an invariant independent of the mechanisms that realize it.

Overall, heap decomposition inference addresses a fundamental goal in programming languages – promoting local reasoning – with a broad range of benefits in optimizing, debugging, verifying, securing, and understanding programs. Thread locality – a property crucial for optimizing multithreaded programs (*e.g.* [1,13]) – can be viewed as decomposing heap objects into scopes defined by threads. Objects in the same scope may signify the likelihood of co-use, and co-allocating them may lead to reduced cache misses, a boon for performance [15] and energy efficiency [33]. Localizing the use of objects can also aid bug localization [31] and analyze the impact of change [4]. Progress in separation logic (*e.g.*[12]) demonstrates the usefulness of locality information in verification. In security, the scope of capabilities [9] – object references in object-oriented programs – is crucial in constructing access control and authority isolation [27,23]. The compositional view of the heap is also beneficial in reverse engineering [16].

This paper makes the following contributions:

- a unified framework to reason about HD and its properties (Sec. 2).
- a novel constraint-based type inference to abstract object access in the presence of the compositional heap as linear constraints (Sec. 4).
- a novel application of linear programming to improve inference precision, guided by the Cypress Principle (Sec. 5).
- a prototyped implementation that analyzes real-world programs (Sec. 6).

2 Hierarchical Decomposition

In this section, we define a unified framework for HD, with three common forms of HD – lexical decomposition, dynamic heap decomposition, static heap decomposition –

[1] A cypress – especially its subspecies known as Mediterranean Cypress – is a tree usually "tall and skinny" in shape. It is a common image, such as in Vincent van Gogh's *Starry Night*.

definition \ form	lexical decomposition ($\ell = \mathtt{x}$)	dynamic heap decomposition ($\ell = \mathtt{d}$)	static heap decomposition ($\ell = \mathtt{s}$)
reasoning about	program	run-time state	program (and its relationship with run-time states)
accessors ($a^\ell \in A^\ell$)	code blocks	objects	abstract objects
accessees ($b^\ell \in B^\ell$)	variable names	objects	abstract objects
elements ($c^\ell \in C^\ell$)	$A^\ell \cup B^\ell$		
access relation $a^\ell \hookrightarrow^\ell b^\ell$	a^ℓ immediately encloses b^ℓ use	heap stores/stack frames of a^ℓ refers to b^ℓ	fields/methods of a^ℓ refers to b^ℓ
scope relation $a^\ell \blacktriangleright^\ell c^\ell$	a^ℓ immediately encloses c^ℓ definition	inferred in standard object model (or induced via language design)	
$\mathtt{ehd}^\ell(\hookrightarrow^\ell, \blacktriangleright^\ell)$ for soundness	necessary and sufficient		necessary

Fig. 1. From Lexical Decomposition to Heap Decomposition

summarized in Fig. 1. Metavariables introduced in this section routinely use superscript labels $\mathtt{x}$, $\mathtt{d}$, $\mathtt{s}$ to differentiate the three forms.

Throughout the paper, we say relation *Rel* : *SetA* $\times$ *SetB* is a *rooted tree relation* if *Rel* is injective, surjective, irreflexive, acyclic, and contains one unique element *elmtA* such that *elmtA* $\in$ *SetA* and *elmtA* $\notin$ *SetB* and $(elmtA, elmtB) \in Rel$ for some *elmtB*. We further say *elmtA* is the *root* of *Rel*. The rooted tree relation intuitively corresponds to the edge set of graph-theoretic rooted directed trees.

2.1 The Essence of Hierarchical Decomposition (though Lexical Decomposition)

Let us start with the most well-understood HD mechanism, block-based lexical scoping. Given a program P, we abstract it as a 5-tuple $\langle A^\mathtt{x}; B^\mathtt{x}; a^\mathtt{x}_\mathtt{G}; \hookrightarrow^\mathtt{x}; \blacktriangleright^\mathtt{x} \rangle$ configuration where

- Accessor/Scope set $A^\mathtt{x}$: the set of code blocks (IDs) in P
- Accessee set $B^\mathtt{x}$: the set of variables (names) in P
- Global scope $a^\mathtt{x}_\mathtt{G}$: the code block that implicitly encloses the entire program
- Access relation $\hookrightarrow^\mathtt{x}$: $A^\mathtt{x} \cup \{a^\mathtt{x}_\mathtt{G}\} \times B^\mathtt{x}$: $a^\mathtt{x} \hookrightarrow^\mathtt{x} b^\mathtt{x}$ says $a^\mathtt{x}$ accesses $b^\mathtt{x}$. Concretely, it is defined as code block $a^\mathtt{x}$ immediately encloses the use of variable $b^\mathtt{x}$.

– Scope relation $\blacktriangleright^{\mathtt{x}}$: $A^{\mathtt{x}} \cup \{a^{\mathtt{x}}_{\mathtt{G}}\} \times A^{\mathtt{x}} \cup B^{\mathtt{x}}$: a rooted tree relation with root $a^{\mathtt{x}}_{\mathtt{G}}$ and $a^{\mathtt{x}} \blacktriangleright^{\mathtt{x}} c^{\mathtt{x}}$ says $a^{\mathtt{x}}$ is the *lexical scope* of $c^{\mathtt{x}}$. Concretely, it is defined as block $a^{\mathtt{x}}$ immediately encloses $c^{\mathtt{x}}$ definition, as a variable declaration or a nested block.

It is important to realize that for any program subject to lexical scoping, a latent invariant must hold between $\hookrightarrow^{\mathtt{x}}$ and $\blacktriangleright^{\mathtt{x}}$. Indeed, this invariant is a concrete instance of a more general invariant that epitomizes hierarchical decomposition:

Definition 1 (The Essence of Hierarchical Decomposition). *HD mechanisms maintain the following invariant between the access relation $\hookrightarrow^{\ell}$ and scope relation $\blacktriangleright^{\ell}$:*

$$\mathtt{ehd}^{\ell}(\hookrightarrow^{\ell}, \blacktriangleright^{\ell}) \stackrel{\text{def}}{=} a_1^{\ell} \blacktriangleright^{\ell} b^{\ell} \wedge a_2^{\ell} \hookrightarrow^{\ell} b^{\ell} \implies a_1^{\ell} \blacktriangleright^{\ell}_{*} a_2^{\ell}$$

where ℓ is the identifier of the HD mechanism itself (such as $\mathtt{x}$), and $\blacktriangleright^{\ell}_{*}$ is the reflexive and transitive closure of $\blacktriangleright^{\ell}$. The invariant can be interpreted in two equivalent ways:

Accessors from Inner Scopes. If a_1^{ℓ} is the scope of b^{ℓ}, then any access to b^{ℓ} must come from an *inner scope* of a_1^{ℓ}, *i.e.* a scope that can be reached from a^{ℓ} according to $\blacktriangleright^{\ell}$.

Accessees in Outer Scopes. If a_2^{ℓ} accesses b^{ℓ}, then the scope of b^{ℓ} must be an *outer scope* of a_2^{ℓ}, *i.e.* a scope that can reach a_2^{ℓ} according to $\blacktriangleright^{\ell}$.

The **Accessors from Inner Scopes** interpretation demonstrates the key benefit of HD: promoting local reasoning. In Sec. 4, the **Accessees in Outer Scopes** interpretation will help us encode the essence of HD in a type inference algorithm. With this formulation, the common notion of lexical scoping is:

Definition 2 (Lexical Decomposition). *Configuration $\langle A^{\mathtt{x}}; B^{\mathtt{x}}; \mathrm{a}^{\mathtt{x}}_{\mathtt{G}}; \hookrightarrow^{\mathtt{x}}; \blacktriangleright^{\mathtt{x}} \rangle$ conforms to lexical decomposition iff* $\mathtt{ehd}^{\mathtt{x}}(\hookrightarrow^{\mathtt{x}}, \blacktriangleright^{\mathtt{x}})$.

Our goal is to use the familiar lexical decomposition to shed light on heap decomposition. As it turns out, lexical decomposition and heap decomposition bear remarkable similarity: they share the same $\mathtt{ehd}^{\ell}$ invariant. For instance, the two interpretations above resonate with deep ownership and owners-as-dominators [5] in ownership types.

2.2 Dynamic Heap Decomposition

The main difference between lexical decomposition and heap decomposition is that the latter is concerned with the access relation and the scope relation *among objects*. Let us first consider the case of dynamic heap decomposition, *i.e.,* HD for the run-time state at a program execution step. For our discussion here, we abstract the run-time state as a configuration $CF^{\mathtt{d}}$ in the form of a 4-tuple $\langle O^{\mathtt{d}}; o_{\mathtt{G}}; \hookrightarrow^{\mathtt{d}}; \blacktriangleright^{\mathtt{d}} \rangle$ including:

– Object set (unified accessor/accessee/scope set) $O^{\mathtt{d}}$: the set of objects (IDs)
– Global scope $o_{\mathtt{G}}$: the implicit "global" object that encloses the bootstrapping code
– Access relation $\hookrightarrow^{\mathtt{d}}$: $O^{\mathtt{d}} \cup \{o_{\mathtt{G}}\} \times O^{\mathtt{d}}$, where $o_1 \hookrightarrow^{\mathtt{d}} o_2$ says o_1 accesses o_2. Under concrete dynamic semantics with heaps and stacks, one possible $\hookrightarrow^{\mathtt{d}}$ relation is the object reference relation[2], where $o_1 \hookrightarrow^{\mathtt{d}} o_2$ holds iff the heap store for object o_1 or any stack frame for any method invocation of o_1 contains a reference to o_2.

[2] Defining the access relation as the reference relation is similar to the conventional capability model [9]. There are alternative ways to define the access relation, such as the accessor reads from and writes to a field of the accessee.

– Scope relation $\blacktriangleright^{\mathtt{d}}$: $O^{\mathtt{d}} \cup \{o_{\mathtt{G}}\} \times O^{\mathtt{d}}$, a rooted tree relation where $o_1 \blacktriangleright^{\mathtt{d}} o_2$ says o_1 is the *dynamic object scope* of o_2, and $o_{\mathtt{G}}$ is the root

To strictly mirror the definition we used for lexical decomposition, the 4-tuple above is indeed a degenerate 5-tuple $\langle A^{\mathtt{d}}; B^{\mathtt{d}}; o_{\mathtt{G}}; \hookrightarrow^{\mathtt{d}}; \blacktriangleright^{\mathtt{d}} \rangle$ where $A^{\mathtt{d}} = B^{\mathtt{d}} = O^{\mathtt{d}}$. In other words, objects are both accessors and accessees in dynamic heap decomposition. With this alignment, the essence of dynamic heap decomposition can be analogously defined:

Definition 3 (Dynamic Heap Decomposition). *Configuration* $\langle \mathrm{O}^{\mathtt{d}}; o_{\mathtt{G}}; \hookrightarrow^{\mathtt{d}}; \blacktriangleright^{\mathtt{d}} \rangle$ *is a dynamic heap decomposition iff* $\mathtt{ehd}^{\mathtt{d}}(\hookrightarrow^{\mathtt{d}}, \blacktriangleright^{\mathtt{d}})$.

Even though the invariant behind lexical decomposition and dynamic heap decomposition is in essence identical, there is often a difference on how the invariant is applied. In the former, both $\hookrightarrow^{\mathtt{x}}$ and $\blacktriangleright^{\mathtt{x}}$ are readily available during parsing, so $\mathtt{ehd}^{\mathtt{x}}$ is mostly used to provide a boolean answer: does the program conform (to lexical decomposition)? In dynamic heap decomposition however, only $\hookrightarrow^{\mathtt{d}}$ is available in Java-like run-times, so $\mathtt{ehd}^{\mathtt{d}}$ is more often applied to *dynamic heap decomposition inference*: given a run-time state whose access relation is $\hookrightarrow^{\mathtt{d}}$, find a $\blacktriangleright^{\mathtt{d}}$ such that $\mathtt{ehd}^{\mathtt{d}}(\hookrightarrow^{\mathtt{d}}, \blacktriangleright^{\mathtt{d}})$.

In plain words, dynamic heap decomposition inference is aimed at (figuratively) organizing the objects of a run-time state into a hierarchy that conforms to the principle of HD, based on information of inter-object access at the run-time state. Just as lexical decomposition, dynamic heap decomposition also promotes local reasoning: it attempts to provide a scope to every object instead of assuming all as "global." Existing work on dynamic UML composition inference (*e.g.* [16]) and dynamic ownership inference (*e.g.* [28]) are concrete instances in this category. In those contexts, the object that forms the "dynamic object scope" of another is called a *compositional object* or an *owner*.

2.3 Static Heap Decomposition

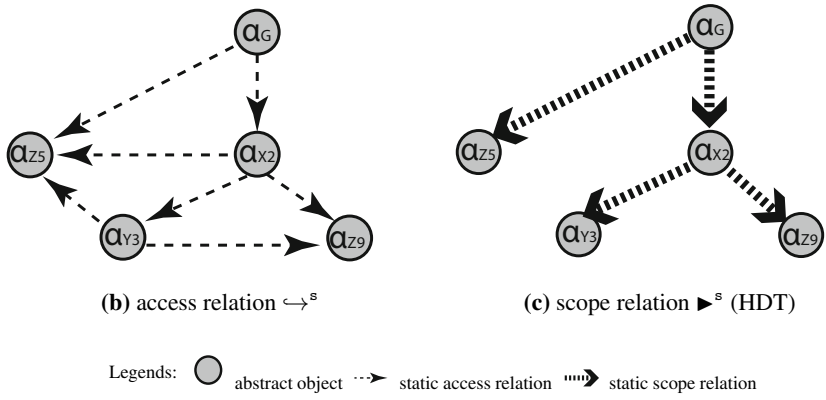
The problem with dynamic heap decomposition is it only applies HD to one particular runtime state of a program's execution (or a finite combination of them). What is more useful is to characterize how heap decomposition can be applied to all run-time states of a program P. This goal is achieved by static heap decomposition, a conservative approximation of dynamic heap decomposition for all run-time states.

Static heap decomposition (checking or inference) systems are constructed over a program abstraction which we represent as configuration $CF^{\mathtt{s}}$, a tuple $\langle O^{\mathtt{s}}; \alpha_{\mathtt{G}}; \hookrightarrow^{\mathtt{s}}; \blacktriangleright^{\mathtt{s}} \rangle$ including:

– Abstract object set (unified accessor/accessee/scope set) $O^{\mathtt{s}}$: the set of static approximations of distinct run-time objects of the program
– Global scope $\alpha_{\mathtt{G}}$: the approximation of the implicit "global" object that encloses the bootstrapping code
– Access relation $\hookrightarrow^{\mathtt{s}}$: $O^{\mathtt{s}} \cup \{\alpha_{\mathtt{G}}\} \times O^{\mathtt{s}}$, where $\alpha_1 \hookrightarrow^{\mathtt{s}} \alpha_2$ says α_1 is the static approximation of an object that accesses an object of which α_2 is the static approximation. We also informally say α_1 *statically accesses* α_2.
– Scope relation $\blacktriangleright^{\mathtt{s}}$: $O^{\mathtt{s}} \cup \{\alpha_{\mathtt{G}}\} \times O^{\mathtt{s}}$, a rooted tree relation where $\alpha_1 \blacktriangleright^{\mathtt{s}} \alpha_2$ says α_1 is the *static object scope* of α_2, and $\alpha_{\mathtt{G}}$ is the root

```
1  class Main {
2    void main() {
3      X@αX1 x;
4      Z@αZ1 z;
5      x = new X@αX2();
6      z = x.mdx();
7    }
8  }
9  class X {
10   Z@αZ2 mdx() {
11     Y@αY1 y1;
12     Y@αY2 y2;
13     Z@αZ3 z1;
14     Z@αZ4 z2;
15     y1 = new Y@αY3();
16     y2 = y1;
17     z1 = new Z@αZ5();
18     z2 = y2.mdy(z1);
19     return y2.fdz;
20   }
21 }
22 class Y {
23   Z@αZ6 fdz;
24   Z@αZ7 mdy(Z@αZ8 z) {
25     this.fdz = z;
26     return new Z@αZ9();
27   }
28 }
29 class Z {...}
```

(a) an example



(b) access relation $\hookrightarrow^{s}$ **(c)** scope relation $\blacktriangleright^{s}$ (HDT)

Fig. 2. Static Heap Decomposition Inference

In the rest of the presentation, we call each object static approximation an *abstract object*, or simply *object* when no confusion can arise. Due to the standard feature of aliasing in object-oriented languages, static approximations may or may not represent distinct run-time objects. We liberally use the term "abstract object" – and its representing metavariable α – to refer to both. For convenience, we also have α subsumes $\alpha_{\mathtt{G}}$.

Following the analogy of dynamic heap decomposition, it would be natural to provide the following definition:

Definition 4 (Static Heap Decomposition). *Configuration* $\langle \mathrm{O}^{s}; \alpha_{\mathtt{G}}; \hookrightarrow^{s}; \blacktriangleright^{s} \rangle$ *is a static heap decomposition iff* $\mathtt{ehd}^{s}(\hookrightarrow^{s}, \blacktriangleright^{s})$.

Let us consider a Java program in Fig. 2(a) to gain some intuition. For convenience, we directly associate each object type declaration with an abstract object ID. For instance, expression **new** $\texttt{Y@}\alpha_{\texttt{Y3}}\texttt{()}$ at Line 15 is the Java expression **new** `Y()` with abstract object $\alpha_{\texttt{Y3}}$. In other words, we treat each program element associated with a type declaration – a local variable, a method parameter, a return value, or an instantiated object – as an abstract object. These abstract objects may or may not lead to "distinct approximations" due to aliasing, but the ones associated with the **new** expressions indeed do, *i.e.*, $O^{\texttt{s}} = \{\alpha_{\texttt{X2}}, \alpha_{\texttt{Y3}}, \alpha_{\texttt{Z5}}, \alpha_{\texttt{Z9}}\}$. $\hookrightarrow^{\texttt{s}}$ can be concretely defined as the points-to information, which in the example is illustrated in Fig. 2(b). One possible $\blacktriangleright^{\texttt{s}}$ is illustrated in Fig. 2(c). It can be easily seen that the two conform to static heap decomposition: every accessor of an object is from inner scopes and every accessee of an object is within outer scopes.

Both checking and inference systems can be constructed for static heap decomposition. A static heap decomposition checker answers whether $\texttt{ehd}^{\texttt{s}}$ holds given a program and its information on $\hookrightarrow^{\texttt{s}}$ and $\blacktriangleright^{\texttt{s}}$ (explicitly or implicitly defined). A *static heap decomposition inference* finds a $\blacktriangleright^{\texttt{s}}$ that satisfies $\texttt{ehd}^{\texttt{s}}$ given a program and its information on $\hookrightarrow^{\texttt{s}}$ (explicitly or implicitly defined). Broadly construed, the typechecking process of ownership type systems [29,6,5] is an instance of static heap decomposition checker. In that context, $\blacktriangleright^{\texttt{s}}$ is the relationship among ownership type parameters, whereas $\hookrightarrow^{\texttt{s}}$ and $\texttt{ehd}^{\texttt{s}}$ are implict in the typechecking rules. As another example, ownership inferences based on points-to analyses [24,22] are instances of static heap decomposition inference. They explicitly compute $\hookrightarrow^{\texttt{s}}$ and finds the $\blacktriangleright^{\texttt{s}}$ that conforms to $\texttt{ehd}^{\texttt{s}}$.

Cypress is a static heap decomposition inference. To highlight the central role of the $\blacktriangleright^{\texttt{s}}$ relation in this context – it is the output of the inference – we reinstate it with the following definition (note that $\blacktriangleright^{\texttt{s}}$ by definition is a tree relation):

Definition 5 (Heap Decomposition Tree (HDT)). *We call* $\blacktriangleright^{\texttt{s}}$ *a* heap decomposition tree *for* P *if configuration* $\langle \mathrm{O}^{\texttt{s}}; \alpha_{\texttt{G}}; \hookrightarrow^{\texttt{s}}; \blacktriangleright^{\texttt{s}} \rangle$ *is a static heap decomposition for* P *for some* $\mathrm{O}^{\texttt{s}}$*,* $\alpha_{\texttt{G}}$*, and* $\hookrightarrow^{\texttt{s}}$.

In the rest of the discussion, we will liberally interpret HDT in a graph-theoretic manner. For instance, the *nodes* of an HDT $\blacktriangleright^{\texttt{s}}$ are the union of the domain and the range of $\blacktriangleright^{\texttt{s}}$, and the *edges* are the set interpretation of $\blacktriangleright^{\texttt{s}}$ itself.

2.4 Challenges

Challenge I: Soundness. Is every static heap decomposition according to Def. 4 a "good" one? At the beginning of Sec. 2.3, we explained a key motivation of constructing static heap decomposition is to "characterize how heap decomposition can be applied to all run-time states of a program." Def. 4 unfortunately does not correspond with the run-time states. For this latter goal, let us first introduce *correspondence relation* $CF^{\texttt{d}} \xrightarrow{\mathcal{X}} CF^{\texttt{s}}$, which says $CF^{\texttt{s}}$ is a static approximation for run-time configuration $CF^{\texttt{d}}$ over *abstraction mapping* $\mathcal{X}$. An abstraction mapping is a simple mapping from o's to α's, where $\mathcal{X}(o)$ is the abstract object approximating o. A correspondence relation $\langle O^{\texttt{d}}; o_{\texttt{G}}; \hookrightarrow^{\texttt{d}}; \blacktriangleright^{\texttt{d}} \rangle \xrightarrow{\mathcal{X}} \langle O^{\texttt{s}}; \alpha_{\texttt{G}}; \hookrightarrow^{\texttt{s}}; \blacktriangleright^{\texttt{s}} \rangle$ is well-formed iff

- $O^{\mathtt{d}} \subseteq \mathtt{dom}(\mathcal{X})$
- $O^{\mathtt{s}} = \mathtt{ran}(\mathcal{X})$
- $\mathcal{X}(o_{\mathtt{G}}) = \alpha_{\mathtt{G}}$
- $\mathcal{X}(o) \hookrightarrow^{\mathtt{s}} \mathcal{X}(o)$ for any $o \hookrightarrow^{\mathtt{d}} o'$
- $\mathcal{X}(o) \blacktriangleright^{\mathtt{s}} \mathcal{X}(o)$ for any $o \blacktriangleright^{\mathtt{d}} o'$

where $\mathtt{dom}(\mathcal{X})$ and $\mathtt{ran}(\mathcal{X})$ operators compute the domain and range of $\mathcal{X}$ respectively.

Definition 6 (Soundness). *Given a program P, its static heap decomposition* $\mathrm{CF}^{\mathtt{s}}$ *is sound with respect to $\mathcal{X}$ iff there exists a* $\mathrm{CF}^{\mathtt{d}}$ *for any run-time state of P, such that* $\mathrm{CF}^{\mathtt{d}} \xrightarrow{\mathcal{X}} \mathrm{CF}^{\mathtt{s}}$ *and* $\mathrm{CF}^{\mathtt{d}}$ *is a dynamic heap decomposition.*

In practice, the $\hookrightarrow^{\mathtt{s}}$ relation in $CF^{\mathtt{s}}$ represents the points-to information, and the $\hookrightarrow^{\mathtt{d}}$ relation in $CF^{\mathtt{d}}$ represents the reference relation at a particular run-time state. The two are determined once the concrete abstractions of the program and the runtime are given. Consequently, the soundness definition here mainly reveals the relationship between the static scope relation in $CF^{\mathtt{s}}$ and the dynamic scope relation in $CF^{\mathtt{d}}$. In other words, a static scope relation $\blacktriangleright^{\mathtt{s}}$ in a sound static heap decomposition must serve as a "template" for at least one dynamic scope relation $\blacktriangleright^{\mathtt{d}}$ at every run time state such that (1) $\blacktriangleright^{\mathtt{d}}$ is indeed "instantiated" from $\blacktriangleright^{\mathtt{s}}$ (according to the last well-formedness condition of the correspondence relation); and (2) $\blacktriangleright^{\mathtt{d}}$ offers hierarchical decomposition for the run-time state (a condition of the soundness definition).

We are able to show Cypress is a sound static decomposition inference in that any HDT (*i.e.* $\blacktriangleright^{\mathtt{s}}$) it infers forms a sound static decomposition against a static semantics with standard points-to abstraction for $\hookrightarrow^{\mathtt{s}}$ and a dynamic semantics with standard definitions of $\mathcal{X}$ and $\hookrightarrow^{\mathtt{d}}$.

Challenge II: Efficient Heap-Wide Characterization. Cypress computes a surjective relation $\blacktriangleright^{\mathtt{s}}$ over the abstract object set ($O^{\mathtt{s}}$ in $CF^{\mathtt{s}}$) of a program. In other words, the scope of every abstract object is determined. The output of Cypress is thus more expressive than decision procedures that answer "given objects α and α', is α in the scope of α'?" or "what is the scope of object α?" or "what objects does scope α' include?"

It is possible to achieve heap-wide characterization by mechanical pairwise applications of the decision procedure "is α in the scope of α'?" In contrast, Cypress computes the entire $\blacktriangleright^{\mathtt{s}}$ relation through a single instance of linear programming.

Challenge III: Precision. First, a well-formed program – *i.e.* a well-typed program according to Java typechecking – at least has one HDT:

Lemma 1 (HDT Existence). *Given a well-formed program whose abstract object set is* $\mathrm{O}^{\mathtt{s}}$, $\{(\alpha_{\mathtt{G}}, \alpha) \mid \alpha \in \mathrm{O}^{\mathtt{s}}\}$ *is an HDT.*

In other words, this HDT is a trivial "egalitarian" tree where $\alpha_{\mathtt{G}}$ is the scope of all abstract objects. The bad news is that this HDT is the least useful for characterizing heap decomposition: it defaults to the "global heap" view.

HDTs may not be unique for a program. For example, Fig. 2(c) is an HDT for program Fig. 2(a), but so is the egalitarian tree we just described. Now that multiple HDTs

may exist, the more interesting question is the preference over them. If we take the graph-theoretic view of the HDTs, observe the "egalitarian" tree above is the "short and broad" whereas the tree in Fig. 2(c) is relatively "tall and skinny." Indeed, the shorter and broader a tree is, the closer it is to the default view of the global heap, and the less it promotes local reasoning.

Cypress is designed with the *Cypress Principle* as guidance. Intuitively, it says whenever a static heap decomposition inference algorithm is faced with a choice between constructing the HDT broader or taller, it should favor the latter. Formally, let us define the depth of α – denoted as $\texttt{depth}(\alpha, \blacktriangleright^{\mathtt{s}})$ – as the length of the path from $\alpha_{\mathtt{G}}$ to α when $\blacktriangleright^{\mathtt{s}}$ is interpreted graph-theoretically. Thus:

Definition 7 (Cypress Principle). *Given P with two sound static heap decompositions $\langle \mathrm{O}^{\mathtt{s}}; \alpha_{\mathtt{G}}; \hookrightarrow^{\mathtt{s}}; \blacktriangleright_1^{\mathtt{s}} \rangle$ and $\langle \mathrm{O}^{\mathtt{s}}; \alpha_{\mathtt{G}}; \hookrightarrow^{\mathtt{s}}; \blacktriangleright_2^{\mathtt{s}} \rangle$, and $\sum_{\alpha \in \mathrm{O}^{\mathtt{s}}} \texttt{depth}(\alpha, \blacktriangleright_1^{\mathtt{s}}) < \sum_{\alpha \in \mathrm{O}^{\mathtt{s}}} \texttt{depth}(\alpha, \blacktriangleright_2^{\mathtt{s}})$, the Cypress Principle favors the latter.*

The ideal is to find an optimal HDT whose aggregated depth (*i.e.* the sum of depth of α's in the abstract object set) is maximal. Indeed, this is strictly the opposite of the egalitarian tree whose aggregated depth is the minimal. In Sec. 5, we shall see such an optimal HDT can be effectively computed by Cypress through linear programming.

Challenge IV: Scalability. Def. 4 appears to indicate that to compute $\blacktriangleright^{\mathtt{s}}$, one needs full knowledge of $\hookrightarrow^{\mathtt{s}}$. This route is followed by some related work [24,22,26]. Cypress demonstrates a type inference approach can avoid the explicit computation of $\hookrightarrow^{\mathtt{s}}$, hence decoupling the scalability of our algorithm from that of points-to analyses. In addition, Cypress yields linear constraints, further promoting scalability.

3 Abstract Syntax

We develop our inference over a small language whose abstract syntax is defined in Fig. 3. The language is similar to Featherweight Java (FJ) [18], where notation $\overline{\bullet}$ represents a sequence of $\bullet$'s. The most noticeable difference here is expressions are A-Normal, a form commonly used for specifying alias analyses over Java programs (*e.g.*, [25]). In this form, (single) expressions include assignment $\mathtt{x} = \mathtt{y}$, field read $\mathtt{x} = \mathtt{y.fd}$, field update $\mathtt{x.fd} = \mathtt{y}$, method invocation $\mathtt{x} = \mathtt{y.md(z)}$ and object instantiation $\mathtt{x} =$ **new** τ. The use of the A-normal form requires us to explicitly declare local variables in the method definition (see M). Such local variable declarations are represented by a type environment Γ that maps variable names to types. Pre-defined variable **this** is Java's self reference. Pre-defined class name `Object` is the root class. A program P is formed by a sequence of classes $\overline{CL}$, followed by local variable declarations Γ, and the bootstrapping expression e.

FJ function $\texttt{fields}(\mathtt{X})$ computes the sequence of fields for class `X`, in the form of F. FJ function $\texttt{mtype}(\mathtt{md}, \mathtt{X})$ computes the signature of method `md` for class `X`, in the form of $\tau \rightarrow \tau'$ where τ and τ' are the argument/return types respectively. FJ-like function $\texttt{mbody}(\mathtt{md}, \mathtt{X})$ computes the method body of `md` for class `X`, in the form of $\mathtt{x.y}.\Gamma.e$ where `x` is the formal argument name, Γ is the local variable declarations,

P	$::= \langle \overline{CL}; \Gamma; e \rangle$	*program*
CL	$::=$ **class** $\mathtt{X}$ **extends** $\mathtt{Y}$ $\{F\ M\}$	*class*
F	$::= \overline{\tau\ \mathtt{fd}}$	*fields*
M	$::= \overline{\tau\ \mathtt{md}(\tau\ \mathtt{x})\{\Gamma\ e\}}$	*methods*
e	$::= s \mid s; e$	*expression*
s	$::= \mathtt{x} = \mathtt{y} \mid \mathtt{x} = \mathtt{y.fd} \mid \mathtt{x.fd} = \mathtt{y}$ $\mid \mathtt{x} = \mathtt{y.md(z)} \mid \mathtt{x} =$ **new** τ	*single expression*
Γ	$::= \overline{\mathtt{x} \mapsto \tau}$	*type environment*
$\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{U}, \mathtt{V}$	$\in \mathbb{CN} \cup \{\mathtt{Object}\}$	*class name*
$\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{u}, \mathtt{v}$	$\in \mathbb{VAR} \cup \{\mathbf{this}\}$	*variable name*
$\mathtt{fd}$	$\in \mathbb{FN}$	*field name*
$\mathtt{md}$	$\in \mathbb{MN}$	*method name*
τ		*type (see Sec. 4)*

Fig. 3. Abstract Syntax

and e is the expression that constitutes the method body. For convenience, we have an explicitly bound variable $\mathtt{y}$ to store the return value. For a method $\mathtt{Y\ md(X\ x)\{Z\ z; z = x;}$ **return** $\mathtt{z;\}}$ in Java syntax, the method body computed by $\mathtt{mbody}$ is $\mathtt{x.u.}(\mathtt{z} \mapsto \tau).(\mathtt{z} = \mathtt{x}; \mathtt{u} = \mathtt{z})$ where τ is the type form in our system for $\mathtt{Z}$. The definitions of $\mathtt{fields(X)}$, $\mathtt{mtype(md, X)}$, $\mathtt{mbody(md, X)}$ are identical to their FJ counterparts, except that $\mathtt{mbody}$ computes a 4-component method body as we just described. In addition, we reuse $\mathtt{X} <: \mathtt{Y}$ to denote $\mathtt{X}$ is a (nominal) subclass of $\mathtt{Y}$. Relation $<:$ is a partial order, whose definition is identical to that of FJ. We defer the concrete definition of type τ to the next section.

4 Type Inference with Linear Constraints over Walk Indices

This section defines a constraint-based type inference to encode the essence of heap decomposition: given a program P, every solution to the constraints generated for P represents an HDT for P.

Walk Index on HDT. Recall the essence of HD is the invariant between the access relation and the scope relation. **Accessees in Outer Scopes** says that every referred object must belong to an outer scope (including the current one) of the referring object. Fig. 4 illustrates 8 possible examples, where the access relation and the scope relation overlay on each figure, and the root of the scope relation is at the top. It is not difficult to observe that the first 6 examples conform to **Accessees in Outer Scopes**, but the last 2 do not.

Graph theoretically, a rooted tree such as the HDT enjoys very strong properties in expressing the relative position of tree nodes. The relative position of α_2 to α_1 can be expressed by the shortest path between them: first "going up" (*i.e.* root-bound) for n_1 edges ($n_1 \geq 0$) from α_1, and then "going down" (*i.e.* leaf-bound) for n_2 ($n_2 \geq 0$) edges until reaching α_2. In short, the relative position can be encoded by a pair of integer values, n_1 and n_2. We say the *walk index* of α_2 from the perspective of α_1 is $\nearrow^{n_1}\searrow^{n_2}$, or

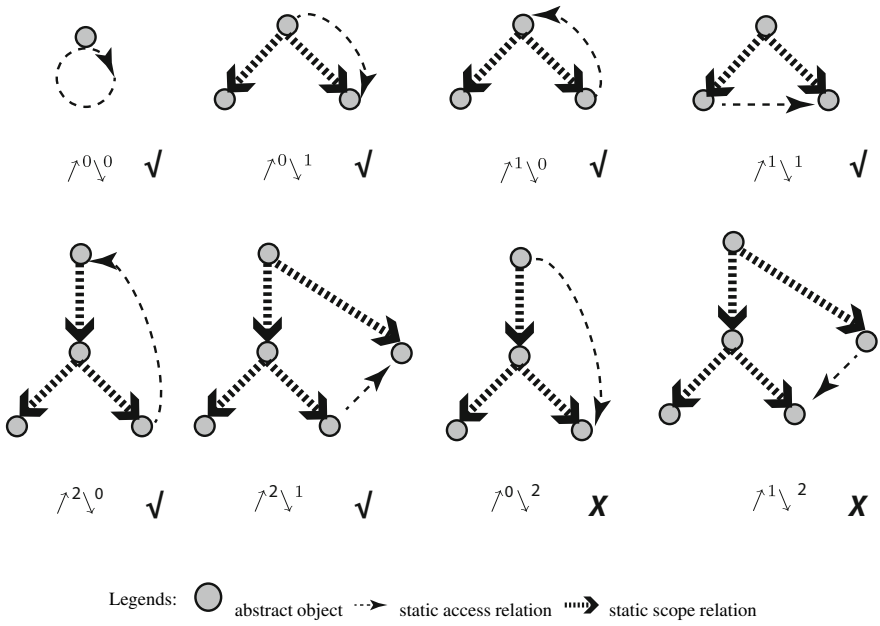


Fig. 4. Access, Scope, and Walk Index

informally say α_1 can walk to α_2 via $\nearrow^{n_1}\searrow^{n_2}$. Encoding "outer/inner scopes" as a pair of integers was first investigated in a previous work by Liu and Smith [21], where the integer pair in a type system design setting was called pedigrees. Broadly, the algorithm introduced in this section can be viewed as a type inference of the type system in that work, an extreme case where every type is inferred.

With the walk index encoding, the essence of HD can be succinctly captured as a restriction on the walk index: the referring object must be able to walk to every of its referred object via walk index $\nearrow^{n_1}\searrow^{n_2}$, *where* $n_2 \leq 1$. Fig. 4 also shows each walk index of the accessee from the perspective of the accessor. The last two examples do not satisfy the requirement of $n_2 \leq 1$, and they violate the principle of HD as well.

Types. In Java-like languages, every referred object is represented by an object reference held by the referring object, and such object reference should have a type in a well-typed Java program. Recall our goal here is to guarantee every referred object to have a walk index (from the perspective of the referring object) that satisfies some restriction ($n_2 \leq 1$ above). Combining the two, we extend the Java type system by building the walk index into the object type, as τ in Fig. 5. Metavariable ω represents a form similar to the walk index we introduced earlier, except that it is associated with a pair of type variables, *p* and *q*, which we call an *up-step type variable* and a *down-step type variable* respectively. The type inference algorithm attempts to solve them with integers (*i.e.* the n_1 and n_2 earlier). We put *p* and *q* into different syntatical categories, so that the constraint solver (Sec. 5) will find non-negative solutions for *p*, whereas 0-1

τ	$::=$	$\mathtt{X}@\alpha[\omega]$	*type*
ω	$::=$	$\nearrow^{p}\searrow^{q}$	*walk index with type variables*
p	$\in$	$\mathbb{PVAR}$	*up-step type variable (solution over* $\{0,1,\dots\}$*)*
q	$\in$	$\mathbb{QVAR}$	*down-step type variable (solution over* $\{0,1\}$ *)*
α	$\in$	$\mathbb{LAB} \cup \{\alpha_{\mathtt{G}}\}$	*abstract object ID*
$\mathcal{K}$	$::=$	$\overline{\kappa}$	*constraint set*
κ	$::=$	$\theta = n \mid \theta \geq n$	*linear constraint*
θ	$::=$	$\theta + t \mid \theta - t$	*linear expression*
t	$::=$	$p \mid q$	*type variable*
n	$\in$	$\{0,1,\dots\}$	*non-negative integer*

Fig. 5. Types and Constraints

binary solutions for q. As we discussed earlier, the latter reflects the principle of HD. When no confusion can arise, we also call ω a walk index.

Overall, a type takes the form of $\mathtt{X}@\alpha[\omega]$, where X is the Java nominal type. For the purpose of presentation, we retain the abstract object ID α in the type. It was used in the example of Fig. 2, where every distinct type declaration is associated with a distinct α. We keep the same convention, except in the case of overriding, we follow the standard FJ requirement of equating the signatures – and hence every component of the types – of the overriding methods and overridden methods. For each program, we define $\mathcal{PV}$ as the smallest mapping that includes (α, p) where $\mathtt{X}@\alpha[\nearrow^{p}\searrow^{q}]$ occurs in the program, and $\mathcal{QV}$ as the smallest mapping that includes (α, q) where $\mathtt{X}@\alpha[\nearrow^{p}\searrow^{q}]$ occurs in the program. We further require $\mathcal{PV}$ and $\mathcal{QV}$ be bijective. In other words, object references with distinct IDs are associated with distinct up-step/down-step type variables.

Let us reiterate the relative nature of the walk index. By definition, it characterizes the referred object from the perspective of the referring object. For instance, if a class X has a field which is associated with type $\mathtt{Y}@\alpha[\omega]$, it says the walk index of any object stored in that field is ω from the perspective of the X object with the field.

Inference Rules. The most critical constraint for achieving HD is that q variables must be binary, which can be succinctly expressed by linear solvers. For a program, these q variables are clearly dependent. We define a type inference to relate type variables (both p and q variables) through linear constraints. Type inference rules are defined in Fig. 6, where $\Gamma \vdash e \backslash \mathcal{K}$ says that expression e yields constraints $\mathcal{K}$ under typing environment Γ. The definition of Γ is given in Fig. 3. Γ, Γ' is defined as the smallest Γ'' such that $\Gamma''(\mathtt{x}) = \Gamma'(\mathtt{x})$ if $\mathtt{x} \in \mathtt{dom}(\Gamma')$ or $\Gamma''(\mathtt{x}) = \Gamma(\mathtt{x})$ if $\mathtt{x} \notin \mathtt{dom}(\Gamma')$ and $\mathtt{x} \in \mathtt{dom}(\Gamma)$. Constraints are defined in Fig. 5. Note that all constraints are linear. We informally say "ω fresh" to mean p and q are fresh type variables where $\omega = \nearrow^{p}\searrow^{q}$.

The key to understanding this set of rules is the four definitions toward the bottom of Fig. 6. The constraint computed by iC is used by (T-New). Combined, it says that the down-step associated with an instantiation must be non-0. From Fig. 4, observe that a walk index whose down-step is 0 means a reference to its object scope (or scope of the scope, and so on). To have an object o instantiate an object o' that represents its scope

for all rules: $\Gamma(\mathtt{x}) = \mathtt{X}@\alpha_{\mathtt{x}}[\omega_{\mathtt{x}}], \Gamma(\mathtt{y}) = \mathtt{Y}@\alpha_{\mathtt{y}}[\omega_{\mathtt{y}}], \Gamma(\mathtt{z}) = \mathtt{Z}@\alpha_{\mathtt{z}}[\omega_{\mathtt{z}}]$

$$\text{(T-New)}\ \Gamma \vdash \mathtt{x} = \textbf{new}\ \mathtt{U}@\alpha[\omega] \backslash (\omega \Rightarrow \omega_{\mathtt{x}}) \cup \mathtt{iC}(\omega)$$

$$\text{(T-Assign)}\ \Gamma \vdash \mathtt{x} = \mathtt{y} \backslash \omega_{\mathtt{y}} \Rightarrow \omega_{\mathtt{x}}$$

$$\text{(T-Write)}\ \frac{\mathtt{U}@\alpha_{\mathtt{u}}[\omega_{\mathtt{u}}]\ \mathtt{fd} \in \mathtt{fields}(\mathtt{X}) \qquad \omega \text{ fresh}}{\Gamma \vdash \mathtt{x.fd} = \mathtt{y} \backslash (\omega_{\mathtt{y}} \Rightarrow \omega) \cup (\omega_{\mathtt{x}} \curvearrowright \omega_{\mathtt{u}} \circlearrowleft \omega)}$$

$$\text{(T-Read)}\ \frac{\mathtt{U}@\alpha_{\mathtt{u}}[\omega_{\mathtt{u}}]\ \mathtt{fd} \in \mathtt{fields}(\mathtt{Y}) \qquad \omega \text{ fresh}}{\Gamma \vdash \mathtt{x} = \mathtt{y.fd} \backslash (\omega \Rightarrow \omega_{\mathtt{x}}) \cup (\omega_{\mathtt{y}} \curvearrowright \omega_{\mathtt{u}} \circlearrowleft \omega)}$$

$$\text{(T-Msg)}\ \frac{\begin{array}{c}\mathtt{mtype}(\mathtt{md}, \mathtt{Y}) = \tau \to \tau' \qquad \tau = \mathtt{U}@\alpha_{\mathtt{u}}[\omega_{\mathtt{u}}] \qquad \tau' = \mathtt{V}@\alpha_{\mathtt{v}}[\omega_{\mathtt{v}}] \\ \omega, \omega', \omega'' \text{ fresh} \qquad \mathtt{mbody}(\mathtt{md}, \mathtt{Y}_i) = \mathtt{u}_i.\mathtt{v}_i.\Gamma_i.e_i \text{ for each } \mathtt{Y}_i <: \mathtt{Y} \\ [\textbf{this} \mapsto \mathtt{Y}_i@\alpha_{\mathtt{y}}[\omega], \mathtt{u}_i \mapsto \tau, \mathtt{v}_i \mapsto \tau'], \Gamma_i \vdash e_i \backslash \mathcal{K}_i \\ \mathcal{K}_{\mathtt{pm}} = (\omega_{\mathtt{z}} \Rightarrow \omega') \cup (\omega_{\mathtt{y}} \curvearrowright \omega_{\mathtt{u}} \circlearrowleft \omega') \\ \mathcal{K}_{\mathtt{rt}} = (\omega'' \Rightarrow \omega_{\mathtt{x}}) \cup (\omega_{\mathtt{y}} \curvearrowright \omega_{\mathtt{v}} \circlearrowleft \omega'')\end{array}}{\Gamma \vdash \mathtt{x} = \mathtt{y.md(z)}) \backslash \mathcal{K}_{\mathtt{pm}} \cup \mathcal{K}_{\mathtt{rt}} \cup \mathtt{thisC}(\omega) \cup \overline{\mathcal{K}}}$$

$$\text{(T-Cont)}\ \frac{\Gamma \vdash s \backslash \mathcal{K} \qquad \Gamma \vdash e \backslash \mathcal{K}'}{\Gamma \vdash s; e \backslash \mathcal{K} \cup \mathcal{K}'}$$

$$\begin{aligned}
\mathtt{iC}(\nearrow^{p} \searrow^{q}) &\stackrel{\text{def}}{=} \{q = 1\} \\
\mathtt{thisC}(\nearrow^{p} \searrow^{q}) &\stackrel{\text{def}}{=} \{p = 0, q = 0\} \\
\nearrow^{p_1} \searrow^{q_1} \Rightarrow \nearrow^{p_2} \searrow^{q_2} &\stackrel{\text{def}}{=} \{p_1 - q_1 = p_2 - q_2, q_2 \geq q_1\} \\
(\nearrow^{p_1} \searrow^{q_1} \curvearrowright \nearrow^{p_2} \searrow^{q_2} \circlearrowleft \nearrow^{p} \searrow^{q}) &\stackrel{\text{def}}{=} \{p = p_1 + p_2 - q_1, q = q_2, p_2 \geq q_1\}
\end{aligned}$$

Fig. 6. Inference Rules and Definitions

directly violates the basic requirement of HD, because references to o (hence access) would exist outside its scope. The constraint computed by `thisC` is used in (T-Msg) for typing **this**. Combined, it says that the walk index of **this** should be solved to $\nearrow^0 \searrow^0$. In figurative terms, an object can walk to itself on the HDT with no steps.

Operator $\omega \Rightarrow \omega'$ constrains the data flow from the object with ω to the object with ω'. Take (T-Assign) for example. The variables in $\omega_{\mathtt{y}}$ and $\omega_{\mathtt{x}}$ are related. Given the q variables binary, the constraints of $\omega_y \Rightarrow \omega_x$ allow for two cases: $\omega_{\mathtt{x}}$ and $\omega_{\mathtt{y}}$ are either solved to identical walk indices, or when $\omega_{\mathtt{y}}$ is solved to $\nearrow^n \searrow^0$, $\omega_{\mathtt{x}}$ can be solved to $\nearrow^{n+1} \searrow^1$. The latter is useful in cases of, say, assigning **this** to a variable defined in the same method. As data flows appear in every expression form, $\Rightarrow$ appears in every rule.

Ternary operator $\omega \curvearrowright \omega' \circlearrowleft \omega''$ computes the constraints for *walk index composition*. Let us imagine we have three objects α, α', α'' where α can walk to α' via walk index ω, α' can walk to α'' via ω', and α can walk to α'' via ω''. Operator $\omega \curvearrowright \omega' \circlearrowleft \omega''$ computes the necessary constraints to keep ω, ω', and ω'' consistent: after all, they are all defined over one HDT. For example in (T-Read), α above is the object enclosing the field read expression. α' is the object represented by `y`, and α'' is the object stored in field `fd` of `y`. The same definition is needed for (T-Write) and (T-Msg). For the latter, imagine one can view parameter passing as a write (to a stack variable) on the message receiver, and value returning as a read (from a stack variable) on the message receiver. Defining $\omega \curvearrowright \omega' \circlearrowleft \omega''$ is purely a graph-theoretic matter on encoding tree path composition, whiich we explain more in the Appendix.

Definition 8 (Whole-Program Constraints). *We use $[\![P]\!]$ to denote the constraints for program P. It is defined as $\mathcal{K}$ where $\Gamma \vdash e \backslash \mathcal{K}$ and $P = \langle \overline{CL}; \Gamma; e \rangle$.*

Optimizations. We do not formalize several standard optimization techniques implemented by our compiler. First, the formal system only requires α (and hence its associated p and q) be distinct for distinct type declarations in the source code. This is known as a 0CFA formulation in polymorphic type inference, or context-insensitive formulation in program analysis. Our compiler refines α as a pair including both the context label and the program point (for the type declaration), and two α's are distinct when either their context labels or program points are different. Second, in (T-Msg), we conservatively consider the method body of every subclass of the message receiver's declared class. In our implementation, a concrete class analysis is in place, so that only the classes whose instances may flow into the message receiver are inspected.

5 Computing and Grooming Heap Decomposition Tree

We now study the solutions to linear constraints in $[\![P]\!]$. We describe how these solutions can be used to compute the scope relation (*i.e.* construct an HDT), its soundness, and how the Cypress Principle can "groom" HDTs to a shape that reflects the principle of HD with optimality. The notion of "grooming" is achieved by preferring an HDT through linear programming. As we shall see, linear programming not only brings efficiency, but also allows us to express HD-related goals as objective functions.

We first introduce some basic notations on constraint solving. We use *placement* $\zeta : \mathbb{PVAR} \rightharpoonup \{0, 1, \dots\} \cup \mathbb{QVAR} \rightharpoonup \{0, 1\}$ to refer to the solution (sub-)vector of a linear constraint set. Recall in Sec. 4, we discussed that the solution to the up-step variable must be a non-negative integer, whereas the solution to the down-step variable must be 0-1 binary. We use predicate $\zeta \downarrow \mathcal{K}$ to denote ζ is a solution to $\mathcal{K}$, defined as every $\kappa \in \mathcal{K}[\zeta]$ is an arithmetic tautology, where $\mathcal{K}[\zeta]$ is standard constraint set substitution. Selection operator $\zeta \uparrow^n$ computes the set of up-step variables mapped to n in ζ. Formally, $\zeta \uparrow^n \stackrel{\text{def}}{=} \{p \mid p \mapsto n \in \zeta\}$.

5.1 From Linear Constraints to HDT

Intuitively, solving the constraints of a program P subsumes resolving all p and q variables associated with the **new** expressions in the program – such as $\alpha_{\texttt{X2}}$, $\alpha_{\texttt{Y3}}$, $\alpha_{\texttt{Z5}}$, $\alpha_{\texttt{Z9}}$

in Fig. 2(a) – to integers. With that, the relative position between the object instantiated by a **new** expression and the object whose class lexically encloses the **new** expression is known. Assuming all instantiation points are reachable from the bootstrapping code, an HDT can be constructed with these relative positions. We formalize this intuition in this section. From now on, since we are mainly concerned with instantiated points, we consider the abstract object IDs associated with instantiated points belong to set $\mathbb{ILAB}$, a subset of $\mathbb{LAB}$. We define convenience function $\mathtt{iA}(P)$ to enumerate all α's in program P where $\alpha \in \mathbb{ILAB}$. We further define $\mathtt{iP}(P)$ as $\{\mathcal{PV}(\alpha) \mid \alpha \in \mathtt{iA}(P)\}$.

For a program P, we define a simple *instantiation relation* – denoted as $\searrow_P$, or $\searrow$ for short – where $\alpha \searrow \alpha'$ says the instantiation point of α' is lexically enclosed by methods of α. It is defined as the smallest relation over $\mathbb{ILAB} \cup \{\alpha_{\mathtt{G}}\}$, s.t. (1) $\alpha_{\mathtt{G}} \searrow \alpha$ for every α appearing in the bootstrapping code of P; (2) $\alpha \searrow \alpha'$ for every α' whose instantiation expression appears in methods of abstract object α. We next define a relation that serves as a "candidate" HDT, as follows:

Definition 9 (Scope Candidate Relation). *Given a program P and $\zeta \downarrow [\![P]\!]$, a scope candidate relation of program P with respect to ζ is a smallest relation $\blacktriangleright^{\mathtt{sc}}$: $\mathtt{iA}(P) \cup \{\alpha_{\mathtt{G}}\} \times \mathtt{iA}(P)$ satisfying:*

$$\alpha \searrow_P \alpha' \text{ and } \zeta(\mathcal{PV}(\alpha')) = n \text{ and } \alpha_0 \blacktriangleright^{\mathtt{sc}} \alpha_1 \text{ and } \alpha_1 \blacktriangleright^{\mathtt{sc}} \alpha_2 \text{ and } \ldots, \alpha_{n-1} \blacktriangleright^{\mathtt{sc}} \alpha \implies \alpha_0 \blacktriangleright^{\mathtt{sc}} \alpha'$$

Intuitively, the definition describes an algorithm to "construct" an HDT monotonically. Using informal terms, if ζ resolves a particular walk index to $\nearrow^3\searrow^1$, the definition here attempts to "put" 3 + 1 elements into $\blacktriangleright^{\mathtt{sc}}$, each of which (hopefully) represents an edge in HDT.

Is a scope candidate relation a (static) scope relation? We first demonstrate:

Lemma 2 ($\blacktriangleright^{\mathtt{sc}}$ as Rooted Tree Relation). *If $\blacktriangleright^{\mathtt{sc}}$ is a surjective scope candidate relation of program P with respect to ζ, then $\blacktriangleright^{\mathtt{sc}}$ is a rooted tree relation with root $\alpha_{\mathtt{G}}$.*

Here we are only interested in surjective scope candidate relations. Imagine a degenerate program where $\mathtt{x} = \textbf{new}\ \mathtt{X@}\alpha[\nearrow^p\searrow^q]$ is the only expression in the bootstrapping code. The solution to p is unbound (*i.e.* any non-negative integer). In this case, a surjective scope candidate relation can only be constructed if $\zeta(p)$ is 0. One can view any non-surjective scope candidate relation as the result of attempting to place objects outside the global scope $\alpha_{\mathtt{G}}$. (As we shall see, this morbid case does not have relevance in practice, because they can be avoided by judicious settings of objective functions.)

Lemma 3 ($\blacktriangleright^{\mathtt{sc}}$ Uniqueness). *If $\blacktriangleright^{\mathtt{sc}}_1$ and $\blacktriangleright^{\mathtt{sc}}_2$ are both surjective scope candidate relations of program P with respect to ζ, $\blacktriangleright^{\mathtt{sc}}_1 = \blacktriangleright^{\mathtt{sc}}_2$.*

We provide a conventional definition for $\hookrightarrow^{\mathtt{s}}$ [8] where $\alpha \hookrightarrow^{\mathtt{s}} \alpha'$ is the standard points-to abstraction. With Lem. 2, we are able to demonstrate:

Lemma 4 ($\blacktriangleright^{\mathtt{sc}}$ as Scope Relation/HDT). *Given a program P and a surjective scope candidate relation $\blacktriangleright^{\mathtt{sc}}$ of P with respect to ζ, configuration $\langle \mathtt{iB}(P); \alpha_{\mathtt{G}}; \hookrightarrow^{\mathtt{s}}; \blacktriangleright^{\mathtt{sc}} \rangle$ is a static heap decomposition where $\hookrightarrow^{\mathtt{s}}$ is static access relation of P.*

Thus, a surjective scope candidate relation $\blacktriangleright^{\mathtt{sc}}$ is a *bona fide* scope relation, an HDT, and is the only HDT according to Lem. 3. From now on, we denote this unique scope candidate relation of program P w.r.t. ζ as $\mathtt{HDT}(P, \zeta)$. It is undefined if the scope candidate relation is not surjective.

5.2 Soundness of Cypress

We define a conventional small-step operational semantics [8] where $\hookrightarrow^{\mathtt{d}}$ is the standard reference relation at run time – as summarized in Fig. 1 – and $\mathcal{X}$ is the predictable mapping from objects and their abstract object IDs. With that, we can state the main soundness theorem of Cypress:

Theorem 1 (Soundness). *Given a program P and any ζ such that $\mathtt{HDT}(P, \zeta)$ if defined, $\mathtt{HDT}(P, \zeta)$ is sound.*

This important theorem tells us the HDT computed here – a data structure purely computed statically – *can* characterize dynamic heap decomposition. Also note that we never need to explicitly compute $\hookrightarrow^{\mathtt{s}}$ – the points-to information is implicit in the linear constraints except for the sake of stating Lem. 4 and constructing the proof for the theorem here – providing a solution to address Challenge IV.

5.3 Compositional Objects

In Java-like languages, evaluating the **new** expression yields a reference to the instantiated object, say o. This implies the reference to o can at least be obtained by the object o' whose method lexically encloses the **new** expression. We call o' the "scope ground zero" of o, *i.e.* the scope of o must either be o' or an outer scope of o'. Thus, the problem of determining whether an object o has o' as its scope can be converted as determining whether $[\![P]\!] \cup \{p = 0\}$ has solutions, where the instantiation expression is **new** $\mathtt{X}@\alpha[\nearrow^{p}\searrow^{q}]$. This is a standard problem for linear solvers, or if we phrase it in a slightly different way, a linear programming problem to minimize p over constraint set $[\![P]\!]$ and check whether the solution is 0.

Formally, we define objective function θ as a linear expression in the form of $p_1 + p_2 \cdots + p_k$ for some $k \geq 1$, which can be abbreviated as $\bigoplus_{j=1..k} p_j$. We represent an instance of linear programming to minimize an objective function θ over constraints $\mathcal{K}$ as $\min_{\mathcal{K}} \theta$. If ζ is the solution of the linear programming instance, *i.e.* $\zeta \downarrow \mathcal{K}$, then $\min_{\mathcal{K}} \theta$ is defined as the restriction of ζ to the domain formed by the variables that appear in θ. Thus, the following set computes the *compositional objects* in program P, *i.e.* the object that has its "scope ground zero" as the scope:

$$\{\alpha \mid \mathcal{PV}(\alpha) \in \biguplus_{p \in \mathtt{iP}(P)} (\min_{[\![P]\!]} p \uparrow^{0})\}$$

This strategy however implies we need to apply linear programming $|\mathtt{iP}(P)|$ times, clearly inefficient when the set of $[\![P]\!]$ is large. Cypress instead only performs linear programming *once*, through:

$$\{\alpha \mid \mathcal{PV}(\alpha) \in \underset{[\![P]\!]}{\mathbf{min}} \bigoplus_{p \in \mathtt{ip}(P)} p \Big\uparrow^{0} \}$$

This form is clearly more efficient, but does it produce the same result as the former? We answer it affirmatively, and the root reason is the *compositionality* of placements:

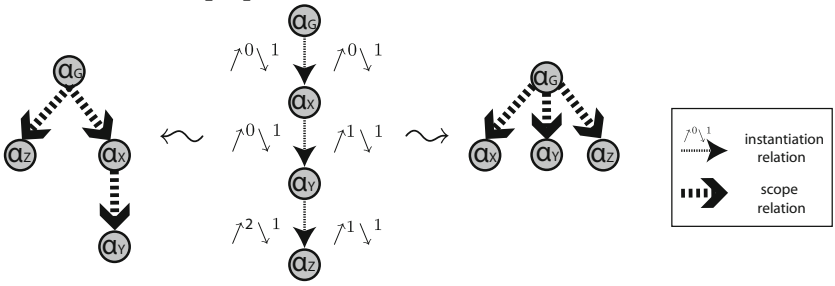
Lemma 5 (Local Placement Compositionality). $\underset{[\![P]\!]}{\boldsymbol{min}}\ \mathrm{p}_i\ = [\mathrm{p}_i \mapsto 0]$ *for* $i = 1, 2$ *iff* $\underset{[\![P]\!]}{\boldsymbol{min}}\ \mathrm{p}_1 + \mathrm{p}_2\ = [\mathrm{p}_1 \mapsto 0, \mathrm{p}_2 \mapsto 0]$ *for any* P *and* $\mathrm{p}_1, \mathrm{p}_2 \in \mathtt{iP}(P)$.

In other words, the second strategy is an optimization of the first.

5.4 Cypress Grooming

The previous section is a primitive use of the Cypress Principle – it favors 0 solutions for the up-step variables of objects, *de facto* placing the object further away from the root. If an object indeed escapes from its scope ground zero, what should be its "minimal escape"? The Cypress Principle guides linear programming for this task, and we term the resulting preferred HDT a *cypress*.

One possible solution is to design a "minimizing all" approach through linear programming, *i.e.* $\underset{[\![P]\!]}{\mathbf{min}} \bigoplus_{p \in \mathtt{ip}(P)} p$. This however may not yield the expected "tall and skinny" cypress. As a counterexample, consider a simple scenario below, where the graph in the center denotes a $\searrow$ relation for a program P, and the label on each edge is the walk index of the head from the perspective of the tail. Observe there might be two different solutions of $[\![P]\!]$ but both satisfy $p + p' + p'' = 2$:



The Cypress Principle – grooming the tree tall and skinny – favors the left tree instead of the right one, but this preference cannot be expressed here. The root of this problem is that the walk index from $\alpha_{\mathtt{x}}$ to $\alpha_{\mathtt{y}}$ and that from $\alpha_{\mathtt{y}}$ to $\alpha_{\mathtt{z}}$ are dependent: a lesser up-step value for an object closer to the root not only helps place itself further away from the root on the HDT, but also helps place the objects it can reach through $\searrow$. Building on this insight, we minimize the sum of the up-steps for each object *relative to the root*. For the example above, note that the (combined) up-steps for $\alpha_{\mathtt{x}}$ relative to $\alpha_{\mathtt{G}}$ is p, and the combined up-steps for $\alpha_{\mathtt{y}}$ relative to $\alpha_{\mathtt{G}}$ is $p + p'$, and the combined up-steps for $\alpha_{\mathtt{z}}$ relative to $\alpha_{\mathtt{G}}$ is $p + p' + p''$. Thus, the objective function to minimize is $(p) + (p + p') + (p + p' + p'')$. Formally, let $\Delta_{P,\alpha}$ denote a set $\{p_1, \ldots, p_n\}$ where

$\alpha_{\texttt{G}} \searrow \alpha_1, \ldots, \alpha_{n-1} \searrow \alpha_n$ and $\mathcal{PV}(\alpha_i) = p_i$ for $i = 1, \ldots, n$. We can now compute the cypress as follows:

Definition 10 (Cypress). *The cypress of program P, denoted as* $\texttt{cypress}(P)$*, is defined as* $\texttt{HDT}(P, \zeta)$*, where*

$$\zeta = \underset{[\![P]\!]}{\boldsymbol{min}} \bigoplus_{\alpha \in \texttt{iA}(P)} \bigoplus_{\texttt{p} \in \Delta_{P,\alpha}} \texttt{p}$$

It would not be difficult to show the scope candidate relation w.r.t. ζ is surjective, and hence $\texttt{HDT}(P, \zeta)$ is defined. We next state an important theorem that says that the cypress we produce is a "tall and skinny" HDT whose aggregated depth is the greatest:

Theorem 2 ("Tall and Skinny"). *Given program P and some ζ such that* $\texttt{HDT}(P, \zeta)$ *is defined,* $\sum_{\alpha \in \texttt{iA}(P)} \texttt{depth}(\alpha, \texttt{HDT}(P, \zeta)) \leq \sum_{\alpha \in \texttt{iA}(P)} \texttt{depth}(\alpha, \texttt{cypress}(P))$.

6 Implementation and Evaluation

A prototype implementation of Cypress has been built on top of the Polyglot framework 2.4 [30]. The compiler supports inheritance, dynamic dispatch, super calls, method/constructor overloading, static fields, static methods, multi-dimensional arrays, inner classes, generics, reflection, autoboxing/unboxing, and enum types. The implementation of Java 1.5 features is modified from a Polyglot extension at UCLA[3]. We choose an open-source linear solver `lpsolve`[4] for constraint solving. Our compiler produces the tree data of the cypress in XML, subsequently rendered by Prefuse[5].

Cypress implements a (field-sensitive) k-object context-sensitive algorithm [25]. To create a stress test for scalability, we choose an expensive form: k in our algorithm is not fixed to a (usually small) constant. The entire chain of instantiation site labels is preserved to represent distinct contexts, except that when recursion happens, we only use the chain of labels between two occurrences of the reappearing label. This technique was used in some compilers previously developed by us [19,7].

Our benchmarks are selected from diverse sources: `puzzle` (PU) [19] solves a famous 4x4 sliding puzzle; `montecarlo` (MO) is from the JavaGrande suite [32]; `jspider` (JS)[6] is an open-source web robot; `messadmin` (ME)[7] is a Java HTTP session monitor; `lusearch` (LU) is from DaCapo[8] suites; `cypress+polyglot` (CY) is our compiler itself. The last benchmark is a meta-circular effort to help us validate the correctness of our implementation. We find this approach very useful for discovering implementation bugs, as we are familiar with the minute details of our own software, and whether the details of its cypress "makes sense" can be quickly examined.

[3] `http://www.cs.ucla.edu/~todd/research/polyglot5.html`
[4] `http://lpsolve.sourceforge.net/`
[5] `http://prefuse.org/`
[6] `http://j-spider.sourceforge.net/`
[7] `http://messadmin.sourceforge.net/`
[8] `http://dacapobench.org/`

name	*#LOC*	*#RM*	$\#I$	$\#F$	$\#V$	$\#EQ$	*GT*(s)	*IT* (s)	*FT* (s)
PU	882	38+1000	314	42+37	572	765	0.49	0.69	0.45
MO	3,128	131+984	294	33+19	206	278	0.48	0.59	0.40
JS	13,986	1434+4329	1761	1557+173	35,134	54,598	45	28.79	24.65
ME	65,356	2144+1973	903	556+191	16,168	22,789	4	6.07	4.01
LU	112,649	6077+2958	1718	657+1086	38,842	72,731	65	39.37	34.84
CY	118,309	4935+14009	6406	4646+1398	107,172	120,185	470	205.98	188.98

Fig. 7. Benchmarking Results (*#LOC*: program LOC; *#RM*: the number of reachable methods (two parts: application methods + library methods); $\#I$: the number of distinct instantiation points; $\#F$: the number of accessed fields (two parts: non-**private** fields + **private** fields); $\#V$: the number of type variables generated by Cypress; $\#EQ$: the number of linear equations generated by Cypress; *GT*/*IT*/*FT*: time, see text for details)

All experiments are conducted on Intel Core Duo 2.53GHz with 4GB RAM, with data reported in Fig. 7. All *#RM*, $\#I$, and $\#F$ data are reported context-sensitively. To see the effect of the stress test we created, note that `cypress+polyglot` reports 6406 instantiation points, significant among the state of the art of program analysis. In comparison, the largest $\#I$ counts are 1261 in [24] and 4152 in [22]. We construct two experiments for validation, one for generating cypresses and the other for finding compositional fields. (The first task subsumes the goal of finding compositional objects.) The last three columns report the time (in seconds) for the two experiments. *GT* reports the time used for all compilation steps other than constraint solving, shared by both experiments. *IT* and *FT* are constraint solving time for the two experiments respectively.

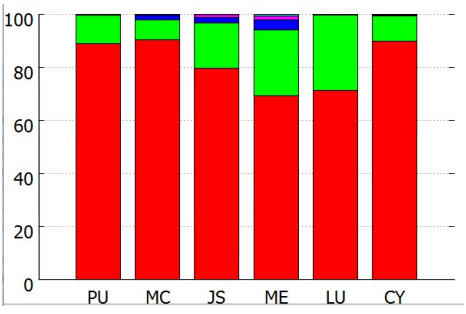


Fig. 8. Distance to "ground zero" (Red : compositional; Green : escape by 1 scope; colors upwards: escape by 2/3/4 scopes)

Encapsulated Objects vs. Escaped Objects. Fig. 8 provides a normalized distribution of the number of distinct instantiation points escaping from "ground zero," computed via Def. 10. Across all benchmarks, the vast majority of the objects (70%-80%) stay "encapsulated" or "owned" within "ground zero." More rigorously, this means any object in this category is never accessed in any outer scope (Def. 1) of the object whose methods or field initializers include the instantiation expression of the former. When objects do not fall into this category, which we intuitively say they are "escaped," it is rare that they escape by more than one scope. This clearly does not result from the lack of scopes in the cypresses – the cypress heights for the benchmarks are 6, 6, 11, 10, 6, 10, respectively. We believe it demonstrates the compositional nature of the heap.

Fig. 9. (Partial) Cypress for PU Benchmark (Best View: Scale to 200%)

Cypress Graphs for Program Understanding. The cypresses of all benchmarks have been rendered in a graphic tree form. Here we include a partial cypress of the smallest benchmark in Fig. 9, showing only the subtree that focuses on application objects (*i.e.* non-library objects). The cypress is rendered with the root on the left, with each edge intuitively says the left-hand node is the scope of the object of right-hand node. More rigorously, there is an edge with α on the left and α' on the right iff $\alpha \blacktriangleright^{\mathtt{s}} \alpha'$.

Each node on the cypress is labeled in the form of $\mathtt{C}\#\mathtt{v}$, where $\mathtt{C}$ is the class name of the represented object, and $\mathtt{v}$ is the abstract object ID to uniquely identify the object. Observe that in a context-sensitive algorithm, it is insufficient to identify an object by class names (or the instantiation program points for that matter). Not presented here, the Cypress tool further accompanies the rendered cypress with a table to associate each $\mathtt{v}$ with context labels – a chain of instantiation program points that ultimately lead to the instantiation of the object represented by $\mathtt{v}$.

The cypress offers a vivid representation of the heap decomposition structure latent in object-oriented programs: how the heap is decomposed from the root, level by

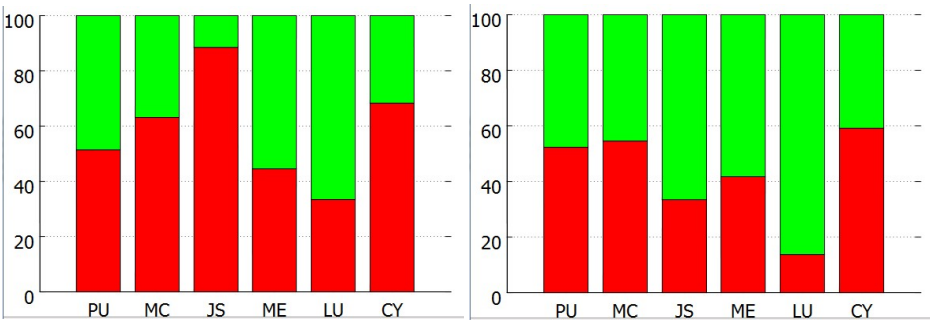
level, as we move from left to right. Observe that the `PuzzleTaskQueue#var2796` object successfully encapsulates its internal linked list representation, an object labeled `ConcurrentLinkedQueue#var2810`. As another example, a `Puzzle#var2709` object encapsulates the `PuzzlePosition#var2729` object, which in turn encapsulates the `Block[]#var2778` array.

The cypresses of all benchmarks are online for review [8]. As the $\#I$ numbers suggest, some of the trees are large.

Compositional Fields. The problem of *compositional fields* is to determine whether a field `fd` of an object α in program P always refers to objects that are never accessed outside the scope formed by α. This is particularly relevant for **private** fields, as a non-compositional **private** field may be counter-intuitive to programmers [22,24,35].

Cypress determines field compositionality as follows. First, observe that each field is declared with a type, which is also associated with a walk index. Let us denote the set of all the up-step variables of such walk indices as $\texttt{fp}(P)$ for program P. If the object stored in the field is local to the scope formed by α, the walk index from α to the stored object must resolve to $\nearrow^0\searrow^1$ or $\nearrow^0\searrow^0$. Following a similar formulation as compositional objects (Sec. 5.3), the set of all compositional fields for program P can be computed by only one instance of linear programming:

$$\{\alpha \mid \mathcal{PV}(\alpha) \in \min_{[\![P]\!]} \bigoplus_{p \in \texttt{fp}(P)} p \Big|^{0} \}$$



(a) private fields **(b)** non-private fields

Fig. 10. Field Compositionality (Compositional? Red / Green : Yes/No)

We next show some experimental results on compositional fields. In Fig. 7, $\#F$ is the number of accessed fields, divided by non-**private** ones and **private** ones. *FT* is the time (in seconds) for finding all composition fields through linear programming. Fig. 10 shows the normalized distribution of field compositionality. Benchmarks are placed from left to right in the same order as in the table earlier. Fig. 10 (a) shows a

phenomenon consistent with previous findings [24,22] – a **private** declaration does not always guarantee compositionality. Our system reports a slightly higher percentage of compositional fields (a difference of 5%-10% on average). Other than the possibility of Cypress being a more precise algorithm, we speculate this may result from our precise settings of context sensitivity thanks to the scalability of linear programming. We also constructed experiments for non-**private** fields, with results in Fig. 10 (b). As expected, the overall percentage of compositional fields has decreased, but interestingly, we found a significant percentage of them are compositional. This might result from the fact that the default modifier choice of Java is not **private**. Programmers uninterested in visibility protection are in fact declaring non-**private** fields.

JATO Support. We have extended Cypress to support language features important for multi-threaded programs – such as representing each thread as a node on the cypress, and differentiating read/write access to fields – and apply the extended tool to assist JATO [20], a system for atomicity enforcement across the JNI boundary. In that extension, any descendent object of the thread node on the cypress is guaranteed to be thread-local according to Java-side analysis. If such an object crosses the boundaries of JNI and its access on the C side falls into some simple patterns (such as only being read/written), the object is statically guaranteed to be thread-local; no locks are needed for their atomicity enforcement. This application of Cypress is a concrete example of a well-established direction: reasoning about thread locality with ownership (*e.g.*[35]).

7 Related Work

The most advanced direction related to our work is ownership type inference, which can be achieved either through dynamic analysis [28,34] or static analysis [10,26,17]. Dynamic approaches are sound only w.r.t. the (finite number of) executions the analysis is performed over, but on the positive side, they can often offer insight into complex structures of programs beyond ownership (an example would be a "butterfly" in [28]). Dietl et. al. designed a tunable static analysis [10] for inferring the modifiers of Universe Types (UT) [11]. One feature of their system is that the inference is reduced to an SAT satisfiability problem, where practical solvers exist. The high-level vision – reducing a non-standard problem into a standard problem – is shared by Cypress, as we reduce our problem to linear programming. As an ownership type system, UT is known to have some distinctive features: it does not conform to deep ownership, but considers a form of interaction between immutability and ownership (known as owners-as-modifiers). As a result, the inference built for UT differs with ours in the underlying invariant. Milanova and Vitek designed dominator inference [26], a type inference for modifiers of Ownership Types (OT) [29,6,5]. OT supports deep ownership, so OT inference is closer in goal to Cypress. Their approach assumes a pre-computation of the points-to set. Huang et. al. [17] designed a unified approach to infer modifiers for both UT and OT. Their system allows the preference over different modifiers to be expressed through a ranking, and an iterative inference is constructed to place priority on higher ranked modifiers. None of the cited systems uses linear programming.

Points-to analyses have been designed to address two related problems: UML composition inference [24] and field composition identification [22]. Explained in our terminology, this line of work first assumes the pre-knowledge of $\hookrightarrow^{s}$ – points-to information from standard points-to analyses – and then (re-)analyzes programs with two tasks: satisfying $\texttt{ehd}^{s}$ and guaranteeing soundness. Cypress demonstrates that a type inference can be constructed for sound static heap decomposition with no need for full knowledge of points-to information. In general, systems in this category may take a very different approach from ours, but they share our goal of constructing a fully automated static analysis.

Language designs for ownership support generally have some inference ability to reduce annotation overhead (*e.g.*[2]), but the goal is not to analyze annotation-free programs. An inference based on Confined Types [14] analyzes unannotated Java code, with a premise that Java packages serve as the scope for objects defined inside the package. Scope Types [3] is an ownership type system lightweight on annotation overhead by design. Our previous work Pedigree Types [21] allows programmers to optionally declare "pedigrees" as type modifiers, a language design incarnation of walk index. Pedigree Types has the ability to infer some type modifiers elided by the programmer. In that light, Cypress considers the extreme case where all pedigree modifiers are ignored – a case Pedigree Types trivially (but unhelpfully) answers "it typechecks" – and demonstrates how linear programming can help precisely recover them. Overall, the route of language design and the route of program analysis for static heap decomposition are complementary.

8 Conclusion

This paper studies heap decomposition, and describes a static ownership type inference that can offer vivid insight into the dynamic nature of software: the compositional view of the runtime heap. In the future, we plan to apply Cypress to two application domains: thread locality for optimization and cache locality for energy efficiency.

Cypress has been implemented as an open-source tool and can be downloaded [8]. The technical report at the same website contains the operational semantics, the proofs, and the cypress graphs of all benchmarks.

Acknowledgments. We thank the reviewers for their thorough and insightful comments, Xiangjin Xu and Ashish Agarwal for their suggestions on linear programming, Scott Smith for earlier discussions on Pedigree Types, Tom Bartenstein for proof reading, and Leena Joseph, Nandan Samant, and Jacob Strohm for developing helper software modules being integrated to our compiler. This work was partially supported by a Google Faculty Award and NSF Award No. CCF-1054515.

References

1. Aldrich, J., Chambers, C., Sirer, E.G., Eggers, S.: Static analyses for eliminating unnecessary synchronization from Java programs. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 19–38. Springer, Heidelberg (1999)

2. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA 2002, pp. 311–330 (2002)
3. Andreae, C., Coady, Y., Gibbs, C., Noble, J., Vitek, J., Zhao, T.: Scoped types and aspects for real-time Java. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 124–147. Springer, Heidelberg (2006)
4. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
5. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales (July 2001)
6. Clarke, D.G., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA, pp. 48–64 (1998)
7. Cohen, M., Zhu, H.S., Emgin, S.E., Liu, Y.D.: Energy types. In: OOPSLA 2012 (October 2012)
8. Cypress, `http://www.cs.binghamton.edu/~davidl/cypress/`
9. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM 9(3), 143–155 (1966)
10. 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)
11. Dietl, W., Müller, P.: Universes: Lightweight ownership for jml. Journal of Object Technology 4(8), 5–32 (2005)
12. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
13. Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: TLDI 2005, pp. 47–58 (2005)
14. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: OOPSLA 2001, pp. 241–255 (2001)
15. Grunwald, D., Zorn, B., Henderson, R.: Improving the cache locality of memory allocation. In: PLDI 1993, pp. 177–186 (1993)
16. Guéhéneuc, Y.-G., Albin-Amiot, H.: Recovering binary class relationships: putting icing on the uml cake. In: OOPSLA 2004, pp. 301–314 (2004)
17. Huang, W., Dietl, W., Milanova, A., Ernst, M.D.: Inference and checking of object ownership. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 181–206. Springer, Heidelberg (2012)
18. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java - a minimal core calculus for Java and GJ. In: TOPLAS, pp. 132–146 (1999)
19. Kulkarni, A., Liu, Y.D., Smith, S.F.: Task types for pervasive atomicity. In: OOPSLA 2010 (October 2010)
20. Li, S., Liu, Y.D., Tan, G.: JATO: Native code atomicity for Java. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 2–17. Springer, Heidelberg (2012)
21. Liu, Y.D., Smith, S.: Pedigree types. In: IWACO 2008, pp. 63–71 (July 2008)
22. Ma, K.-K., Foster, J.S.: Inferring aliasing and encapsulation properties for Java. In: OOPSLA 2007, pp. 423–440 (2007)
23. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web applications. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 125–140 (2010)
24. Milanova, A.: Precise identification of composition relationships for uml class diagrams. In: ASE 2005, pp. 76–85 (2005)
25. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. TOSEM 14, 1–41 (2005)

26. Milanova, A., Vitek, J.: Static dominance inference. In: TOOLS (49), pp. 211–227 (2011)
27. Miller, M.S., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 195–229. Springer, Heidelberg (2005)
28. Mitchell, N.: The runtime structure of object ownership. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 74–98. Springer, Heidelberg (2006)
29. Noble, J., Potter, J., Vitek, J.: Flexible alias protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
30. 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)
31. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39 (2003)
32. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Supercomputing 2001 (2001)
33. Su, C.-L., Despain, A.M.: Cache design trade-offs for power and performance optimization: a case study. In: Proceedings of the 1995 International Symposium on Low Power Design, ISLPED 1995, pp. 63–68. ACM, New York (1995)
34. Wren, A.: Ownership type inference. Master's thesis, Imperial College (2003)
35. 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)

Appendix: Walk Index Composition Constraints

In this section, we offer an explanation to the constraints used to capture walk index composition, as defined by $\omega_1 \curvearrowright \omega_2 \circlearrowleft \omega$ in Fig. 6. The relationship among ω_1, ω_2, ω is the standard problem of tree path composition. Not to lose generality, there are only two subcases, depending on the comparison between the ω_1's down-step (q_1) and ω_2's up-step (p_2):

- (a) If $q_1 \leq p_2$, then $p = p_1 + p_2 - q_1$ and $q = q_2$.
- (b) If $q_1 > p_2$, then $p = p_1$ and $q = q_1 - p_2 + q_2$.

For (b), observe that to satisfy $q_1 > p_2$, any solution ζ must have $\zeta(q_1) = 1$ and $\zeta(p_2) = 0$. In this case, the only way to satisfy $q = q_1 - p_2 + q_2$ is $\zeta(q_2) = 0$. In other words, ω_2 must be solved to $\nearrow^0\searrow^0$, with the only satisfiable object reference being **this**. Note that in Fig. 6, in all uses of $\omega_1 \curvearrowright \omega_2 \circlearrowleft \omega$, ω_2 is a walk index associated with a method formal parameter, a method formal return type, or a field declaration. Allowing them to be inferred with a walk index $\nearrow^0\searrow^0$ is hardly useful for HD inference. The constraints defined for $\omega_1 \curvearrowright \omega_2 \circlearrowleft \omega$ in Fig. 6 only considers (a). (Observe however, **this** can still be passed as a method parameter, method return value, or stored in fields, except that the walk index of the associated formal paramter, return type, or field type would be solved to $\nearrow^1\searrow^1$.)

A Formal Semantics for Isorecursive and Equirecursive State Abstractions

Alexander J. Summers[1] and Sophia Drossopoulou[2]

[1] ETH Zurich
alexander.summers@inf.ethz.ch
[2] Imperial College London
s.drossopoulou@imperial.ac.uk

Abstract. Methodologies for static program verification and analysis often support recursive predicates in specifications, in order to reason about recursive data structures. Intuitively, a predicate instance represents the complete unrolling of its definition; this is the *equirecursive* interpretation. However, this semantics is unsuitable for static verification, when the recursion becomes unbounded. For this reason, most static verifiers differentiate between, e.g., a predicate instance and its corresponding body, while providing a facility to map between the two; this is the *isorecursive* semantics. While this latter interpretation is usually implemented in practice, only the equirecursive semantics is typically treated in theoretical work.

In this paper, we provide both an isorecursive and an equirecursive formal semantics for recursive definitions in the context of Chalice, a verification methodology based on implicit dynamic frames. We show that development of such formalisations requires addressing several subtle issues, such as the possibility of infinitely-recursive definitions and the need for the isorecursive semantics to correctly reflect the restrictions that make it readily implementable. These questions are made more challenging still in the context of implicit dynamic frames, where the use of heap-dependent expressions provides further pitfalls for a correct formal treatment.

1 Introduction

Recursive definitions of a program's state, are widely employed in techniques for program specification, verification and static analysis. Common techniques include recursive predicates, pure methods, abstraction functions and model fields. The ability to express recursion in specifications is needed to describe programs which themselves manipulate recursively-defined data structures, since it is impossible for a specification to explicitly describe each of the locations involved when accessing the structure. For example, a method which computes the sum of the values in a linked-list will need to access a statically-unbounded number of heap locations to do so. To solve this specification problem in the context of permission-based methodologies such as separation logic [8,13], *recursive abstract predicates* [15] were introduced. Predicate definitions can be provided as part of a

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 129–153, 2013.

program's specification, and the meaning of a predicate name is defined in terms of an assertion (the predicate *body*), which may itself include instances of the same predicate. In this way, it is possible for a predicate instance to implicitly require permission to access, e.g., every *next* field in a linked-list. The intuitive meaning of such a predicate symbol is that it represents everything implied by the (recursive) unrolling of its definition; this is the *equirecursive interpretation* of the recursive definition [1].

However, static verifiers (and other tools) cannot predictably reason directly in terms of an equirecursive semantics, since at verification time it is impossible to know when to stop unrolling such a definition. For this reason, many verifiers make use of ghost *fold* and *unfold* operations for handling recursively-defined predicates, which explicitly exchange a predicate name for its body (or vice versa). These operations may be explicitly provided in the source code (e.g., [9,11]), implicitly specified via heuristic rules (e.g., [5]), or tools may try to infer them by other means (e.g., [2]); their eventual role is the same. In the absence of *fold* and *unfold* operations, the information implied by the unrolling of a currently-held predicate instance is not made available to the verifier. Such a treatment of recursive definitions differentiates between holding a predicate instance and holding its body, while providing a means by which the one can be explicitly exchanged for the other; this is the *isorecursive interpretation*.

The critical aspect of an isorecursive semantics is that it can be used as the basis for building static tools, while the equirecursive semantics cannot (without the undesirable possibility of potentially infinitely applying recursive definitions, in a so-called *matching loop* [12]). Nonetheless, an equirecursive semantics is much closer to an intuitive runtime model for a methodology, and theoretical papers which formalise verification logics typically treat recursive definitions in this natural way [16]. This creates a mismatch between the formalised assertion logic semantics, and that typically implemented in tools; one of the aims of this paper is to address this mismatch.

Abstraction functions provide a different mechanism for expressing properties of recursive structures. Function definitions can traverse data structures and return abstract values which summarise the contents in ways which abstract over the underlying representation. Pure methods, as used in specification methodologies such as Eiffel, JML and Spec♯ play a similar role, as do model fields. Formalising such functions in a specification language requires care, since a function which is not well-defined (e.g., $bad() = bad() + 1$) can easily lead to inconsistency in the logic; this issue is complicated by the fact that many useful heap-dependent functions do not always have an obvious termination measure expressible in terms of their arguments. Indeed, a normal *length()* function will typically not terminate for cyclic list structures. Furthermore, the unrolling of function definitions also needs careful control for a static tool to handle them practically.

In this paper, we investigate the isorecursive and equirecursive semantics of recursive specification constructs. Concretely, we base our work on the *implicit dynamic frames (IDF)* specification logic of Smans [21]. This logic has been recently shown [17,18] to have close connections with separation logic, however,

it has the advantage for us of including both recursive predicates and heap-dependent abstraction functions, as well as the ability to express *unfolding expressions* which explicitly "peek" inside recursive definitions; the combination of these features makes the work presented both more challenging and more general. The recursive aspects of IDF have not been given a direct assertion semantics before; we give both isorecursive semantics (suitable as the basis for a verifier) and equirecursive semantics (suitable for comparison with a runtime model, and for proving soundness). We extend both assertion semantics to corresponding Hoare Logics, based on a subset of the Chalice programming language [10], and discuss how our isorecursive model lends itself to implementation, and the related possibility of isorecursive states not having a "real" equirecursive counterpart. We define mappings from the isorecursive model to the equirecursive, and show how the various corresponding concepts are formally related. Finally, we define a novel interleaving operational semantics for our language, and prove soundness of our Hoare Logics.

While we work in the context of IDF, the issues arising regarding recursive definitions are much more generic, and the discussions and solutions presented here can easily be adapted for the formalisation of approaches based on e.g., separation logic, and are relevant for the construction of soundness arguments for other techniques such as decision procedures and static analyses for recursive definitions. One of the goals of this work is to identify and elaborate on the challenges which arise, in order to help other researchers facing them. The development of such formalisations requires addressing several subtle issues, regarding both the possibility of infinitely-recursive definitions and the need for the isorecursive semantics to correctly reflect the restrictions that make it readily implementable. The mismatch between the intuitive (equirecursive) semantics and that implemented in tools can lead to pitfalls in practice; for example, a prototype implementation of recursive definitions in the Chalice verifier was unsound for this reason; a correct solution has only recently been proposed [7].

Contributions. The contributions of this paper are:

- An equirecursive semantics for IDF expressions and assertions, including recursive functions and predicates. This is the first direct assertion semantics for IDF which handles recursive definitions.
- An isorecursive semantics for IDF expressions and assertions; to our knowledge, this is the first such assertion semantics which reflects the distinction between holding a predicate and knowing its body.
- Hoare logics for both approaches; in particular, the isorecursive Hoare Logic includes novel rules for folding and unfolding predicates, and tracking associated information with unfolding expressions.
- Encodings and results which formally relate the two semantics, connecting that used at verification time with that used in soundness proofs.
- A novel operational semantics for Chalice, and a soundness result showing that (isorecursive) verification guarantees runtime soundness. This is the first soundness proof for the Chalice approach including recursive definitions.

A technical report, with auxiliary definitions and proofs is available online [23].

2 Equirecursive Semantics for Predicates and Functions

In this section we define the syntax and semantics of expressions and assertions in the equirecursive setting. Our treatment is based on the work of Parkinson and Summers [17]. Their work did not include any kind of predicates or functions in the assertion language; we address these issues here.

Running Example for the Equirecursive Setting. We will use the following predicate *List*, along with two functions *length* and *bad* as running examples; their meanings are explained in this section (the lookup function *Body* returns the body of the definition of predicates and functions):

$$\begin{array}{ll} \mathit{Body(List)} & \equiv \mathbf{acc}(\mathsf{this}.\mathit{next}) * \mathbf{acc}(\mathsf{this}.\mathit{val}) \\ & \quad *(\mathsf{this}.\mathit{next} \neq \mathsf{null} \rightarrow \mathsf{this}.\mathit{next}.\mathit{List}) \\ \mathit{Body(length)} & \equiv (\mathsf{this}.\mathit{next} = \mathsf{null}\,?\,1 : 1 + \mathit{next.length()}) \\ \mathit{Body(bad)} & \equiv 1 + \mathit{bad()} \end{array}$$

2.1 Implicit Dynamic Frames

Implicit dynamic frames allows expressions which depend on the heap, e.g., in an assertion $x.f.g = \mathsf{this}$. In order to make the meaning of such assertions *robust* to interference from other threads, a notion of *permissions* is employed. Special assertions $\mathbf{acc}(e.f)$ called *accessibility predicates* denote a permission to access the heap location $e.f$, at most one of which is present (per location) in the system at once. Assertions used in specifications must be *self-framing*, which means they include permissions to all heap locations that they dereference. For example, the assertion $x.f.g = \mathsf{this}$ is not self-framing, but $\mathbf{acc}(x.f) * \mathbf{acc}(x.f.g) * x.f.g = \mathsf{this}$ is. The separating conjunction $*$ is related to that of separation logic; it acts just as logical conjunction, but behaves *multiplicatively* with respect to accessibility predicates; that is, $\mathbf{acc}(x.f) * \mathbf{acc}(\mathsf{this}.f)$ requires permission to both locations *separately*, and so it implicitly guarantees that this and x cannot be aliases.

Defining a formal assertion semantics for implicit dynamic frames is challenging. Parkinson and Summers [17] defined a semantics for a core of the logic, and, amongst other questions, addressed the issue of the semantics of assertions which are *not* self-framing. For example, what should be the meaning of $x.f.g = \mathsf{this}$? In a state which does not hold permissions to the two heap locations, evaluation of this expression depends on the other threads, and so giving it a deterministic semantics seems incorrect. However, the difficulty is that a compositional definition of the semantics of assertions cannot "see" whether the appropriate permissions to $x.f$ and $x.f.g$ are held by the current thread. The solution used in [17] is, to give the expression the semantics it should have *assuming* the appropriate permissions are held; i.e., read from the heap regardless. Since all assertions are additionally checked to be self-framing, in the end this means that the above semantics is only applied in the situation in which it makes sense. As we will show in subsection 2.3, similar issues arise when adding recursive definitions to the logic, but their treatment needs to be different.

2.2 Recursive Predicates and Functions

Implicit dynamic frames supports two kinds of recursive definitions in assertions. In this paper, we use the terminology of the Chalice tool [10], and call them *predicates* and *functions*. Predicates can be defined recursively, and their bodies are assertions. Allowing specifications to mention predicates as well as field permissions and boolean expressions, makes it possible for, e.g., a pre-condition to require all permissions to a recursive data structure. For example, the predicate *List* requires permission to all *next* and *val* field locations in a linked list.

Implicit dynamic frames also supports recursively-defined *functions* as part of the syntax of expressions. For example, the assertions this.*length()=4* and this.*itemAt(2)=0* use functions to expose additional information about the internals of a list. Functions typically correspond to the "pure methods" of an implementation[1]. The body of a function is an expression (while the body of a method is a statement, and may have side-effects); function invocations can occur in expressions, including inside specifications, while methods cannot. To avoid the potential for unsoundness, we take care that function definitions *terminate*. For example, the definition of the function *bad* would cause the verifier to deduce inconsistency wherever *bad* were made available.

2.3 Handling Infinite Recursion

The introduction of recursively-defined functions and predicates opens up the potential of non-termination when evaluating assertions. What should be the semantics of a predicate instance whose definition can be unrolled infinitely? And what should be done with function definitions that do not terminate? It is tempting to say that definitions which *may* not terminate should be forbidden. But this would be too restrictive: for example, even though the linked list predicate *List* does not terminate in a heap in which this=this.*next*, such predicate definitions are essential for traversing recursive data structures, and cannot be dispensed with. Similarly, *length*() does not terminate in the case that this= this.*next*. Indeed, there is not even any obvious termination measure in terms of the function's signature that could be used to prove termination of this function definition. Nonetheless, such functions cannot be dispensed with either.

For the equirecursive setting, in which an idealised mathematical semantics is appropriate, an elegant (and reasonably standard) way of handling custom predicate definitions is to make the semantics of infinitely-unrollable predicate instances *false*. We take this approach; that is, we interpret predicate definitions by their least fixed points. In this way, we build into the logic the implicit assumption that all predicate *instances* have finite definitions. Forbidding infinite predicate instances does not harm our expressiveness in practice, since, as will be explained in the next section, such predicate instances could never be obtained in a verifier based on an isorecursive semantics. Thus, any program point at which an infinite predicate instance is required (for example, in a method precondition

[1] Indeed, this is the terminology used in [21].

of the form *List*∗this=this.*next*) is actually unreachable code, and thus it is operationally consistent to assign *false* to such assertions.

Now consider the semantics of potentially-non-terminating functions. The *length* function shows that we must admit function definitions which do not necessarily terminate in *all* states. This means that our assertion semantics needs to cope with the possibility of evaluating a function call whose naïve semantics would cause undefinedness. For example, consider the assertion *List*length()=3*. In the case where this=this.*next* holds, *List* will be false (due to the least fix-point treatment of predicates), and we would like the overall assertion to also mean false. But a naïve definition for expression semantics might give the conjunct in which the function call occurs an ill-defined meaning. To avoid the need for relying on a short-cutting semantics for conjunctions, we define an expression semantics that is *total*, even for naturally non-terminating function calls. We achieve this by the introduction of *error values*, which are dummy values used in place of a non-terminating expression evaluation. Since the overall assertion will be *false* whenever these error values occur, this means that we implement the natural semantics for function calls in all situations where the meaning matters.

2.4 Syntax

Definition 1 (Expressions and Assertions). *We define the syntax of* equi-expressions *(ranged over by e) and* equi-assertions *(ranged over by a) as follows:*

$$\begin{array}{lcl} e & ::= & \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid x \mid e.f \mid e.g(e) \mid e = e \mid (e\,?\,e : e) \\ a & ::= & e \mid \mathbf{acc}(e.f, q) \mid e \to a \mid a * a \mid e.P \mid \mathit{Thread}(x, m, y, z) \end{array}$$

In the above, x, y, z range over program variables, f over field identifiers, g over function names, P over predicate names, m over method names, and q over rational numbers in $(0, 1]$. *The three* reserved variable names, this, X, *and* method, *represent the current receiver, method parameter, and method; the latter may not be used explicitly in expressions, and its role will be explained shortly.*

We implicitly require expressions and assertions to be type-correct, *e.g.* in $e.f$ the type of e should have a field f. Functions have one formal parameter, called X; fewer or more parameters can be encoded. Other logical connectives over equi-expressions, such as $\wedge$, $\vee$ and $\neg$ can be encoded. Note that the implication connective $\to$ is restricted to only allow *expressions* (rather than assertions) on the left-hand side. This restriction is common to most practical verification tools based on separation logic or implicit dynamic frames; it makes it possible to avoid an assertion semantics which needs to quantify over states (see *e.g.* [14,17]); a problematic feature for automatic verification. Similarly, negation is only encodable for boolean expressions. Thus, $\mathbf{acc}(\mathsf{this}.f, 1) * \mathsf{this}.f{=}5$, and $\mathsf{this}.f{=}5 \to \mathbf{acc}(\mathsf{this}.f, 1)$ are assertions according to our definitions, while $\mathbf{acc}(\mathsf{this}.f, 1) \to \mathsf{this}.f = 5$ is not.

The $\mathit{Thread}(x, m, y, z)$ assertion is used to record information about other threads currently running; intuitively, it has the meaning that x stores a *token* (a runtime value representing another thread), and that this thread was forked to

execute method m with receiver y and parameter z. The role of these assertions will be made clear in Section 5; they do not play a significant role with respect to the handling of recursion in our assertion semantics.

2.5 Semantics

As in [17], the semantics of assertions is defined in terms of permission masks π, heaps H, and environments σ. We employ a $*$ connective to combine permissions.

Definition 2 (Notation and Preliminaries).
We assume a set of values consisting of at least true, false, null, *object identifiers (ranged over by ι), thread identifiers (ranged over by t), method names (ranged over by m), and one distinct error value per type* tp *(denoted by* $\mathsf{error}_{\mathsf{tp}}$*).*
Heaps*, ranged over by H, are maps (total functions) from pairs of either object identifier and field name or thread identifier and field name, to values.*
Environments*, ranged over by σ, are maps from variables to values.*
Equi-permission-masks*, ranged over by π, are maps from pairs of either object identifier and field name or thread identifier and field name, to nonnegative values in $\mathbb{Q}$.*
We define the operator $+$ to combine permission masks as follows:
$(\pi + \pi')(\iota, f) = \pi(\iota, f) + \pi'(\iota, f)$.
An equi-permission-mask π is well-formed*, written $\models \pi$, if its range is within $[0, 1]$, i.e., we define $\models \pi$ iff $\forall \iota, f.\ \pi(\iota, f) \in [0, 1]$*
The (overloaded) lookup function Body returns the body of a function (an expression) or of a predicate (an assertion). Predicates have the implicit parameter this*, and functions have the implicit parameters* this*, and* X*.*

An unusual feature above is the inclusion of field locations (in heaps and permission masks) with thread identifiers as receiver. This is in order to permit assertions which track information about other threads; we use three *ghost fields*, called `recv`, `param`, `meth`, which are used to record the receiver, parameter and current method name for a given thread identifier. These fields (which are the only fields defined for a thread identifier) are ghost in the sense that they are not present in runtime heaps (see Section 7 for details). A further novelty in Definition 2 is the error value, $\mathsf{error}_{\mathsf{tp}}$, motivated earlier, and used to define the value of expressions when their evaluation is infinite:

Definition 3 (Value of Equi-Expressions). *The* evaluation *of equi-expressions e to values v in a state consisting of heap H, and environment σ, is defined through a predicate $e \Downarrow_{H,\sigma} v$ as follows* :

$x \Downarrow_{H,\sigma} \sigma(x)$ $\quad \mathsf{null} \Downarrow_{H,\sigma} \mathsf{null}$ $\quad \mathsf{true} \Downarrow_{H,\sigma} \mathsf{true}$ $\quad \mathsf{false} \Downarrow_{H,\sigma} \mathsf{false}$

$e.f \Downarrow_{H,\sigma} v$ $\quad$ *if* $e \Downarrow_{H,\sigma} \iota$ *and* $H(\iota, f) = v$.

$e.g(e') \Downarrow_{H,\sigma} v$ $\quad$ *if* $e \Downarrow_{H,\sigma} \iota$ *and* $e' \Downarrow_{H,\sigma} v'$ *and* $Body(g) \Downarrow_{H,\sigma'} v$, *where* $\sigma' = [(\mathsf{this} \mapsto \iota), (\mathsf{X} \mapsto v')]$.

$e = e' \Downarrow_{H,\sigma} \mathsf{true}$ $\quad$ *if* $e \Downarrow_{H,\sigma} v$ *and* $e' \Downarrow_{H,\sigma} v$ *for some* v.

$e = e' \Downarrow_{H,\sigma} \mathsf{false}$ $\quad$ *if* $e \Downarrow_{H,\sigma} v$ *and* $e' \Downarrow_{H,\sigma} v'$ *and* $v \neq v'$.

$(e_1 \,?\, e_2 : e_3) \Downarrow_{H,\sigma} v$ $\quad$ *if* $e_1 \Downarrow_{H,\sigma} \mathsf{true}$ *and* $e_2 \Downarrow_{H,\sigma} v$.

$(e_1 \,?\, e_2 : e_3) \Downarrow_{H,\sigma} v$ $\quad$ *if* $e_1 \Downarrow_{H,\sigma} \mathsf{false}$ *and* $e_3 \Downarrow_{H,\sigma} v$.

We now define the value *of an expression e in the context of H and σ, as follows*:

$$[\![e]\!]_{H,\sigma} = \begin{cases} v & \text{if } e \Downarrow_{H,\sigma} v \\ \mathsf{error}_{\mathsf{tp}} & \text{if } \nexists v.\ e \Downarrow_{H,\sigma} v \ \text{ and } e \text{ has type } \mathsf{tp}. \end{cases}$$

Note that $\mathsf{error}_{\mathsf{tp}}$ is a value, and cannot be used in source language expressions. Thus, in a configuration H', σ' in which z points to a cycle, we obtain that $[\![z.length()]\!]_{H',\sigma'}$=$\mathsf{error}_{\mathsf{int}}$, and $[\![z.length() = 3]\!]_{H',\sigma'}$= $\mathsf{error}_{\mathsf{bool}}$. We can represent $z.length() \neq 3$ through $(z.length() = 3\,?\,\mathsf{false} : \mathsf{true})$; we would then obtain $[\![z.length() \neq 3]\!]_{H',\sigma'} = \mathsf{error}_{\mathsf{bool}}$. The presence of error values in the above definition may seem surprising, but we will shortly show that such values can only occur when the meaning of the expression is irrelevant to the assertion in which it occurs. First, we define the semantics of equirecursive assertions:

Definition 4 (Semantics of Equi-Assertions). *We define the semantics of equi-assertions in a state comprised of permissions π, heaps H, and environment σ, as the least fixpoint of the following equations:*

$$\begin{array}{lll}
\pi, H, \sigma \models_{\mathsf{E}} e & \iff & [\![e]\!]_{H,\sigma} = \mathsf{true} \\
\pi, H, \sigma \models_{\mathsf{E}} \mathbf{acc}(e.f, q) & \iff & \pi([\![e]\!]_{H,\sigma}, f) \geq q \\
\pi, H, \sigma \models_{\mathsf{E}} e \to a & \iff & ([\![e]\!]_{H,\sigma} = \mathsf{true}) \Rightarrow \pi, H, \sigma \models_{\mathsf{E}} a \\
\pi, H, \sigma \models_{\mathsf{E}} a_1 * a_2 & \iff & \exists \pi_1, \pi_2 : \pi = \pi_1 + \pi_2 \\
 & & \quad \textit{and } \pi_1, H, \sigma \models_{\mathsf{E}} a_1 \textit{ and } \pi_2, H, \sigma \models_{\mathsf{E}} a_2 \\
\pi, H, \sigma \models_{\mathsf{E}} e.P & \iff & \pi, H, \sigma \models_{\mathsf{E}} \mathit{Body}(P)[e/\mathsf{this}] \\
\pi, H, \sigma \models_{\mathsf{E}} \mathit{Thread}(x, m, y, z) & \iff & \pi[\sigma(x), \mathtt{recv}] = \wedge H[\sigma(x), \mathtt{recv}] = \sigma(y) \\
 & & \wedge\ \pi[\sigma(x), \mathtt{param}] = 1 \wedge H[\sigma(x), \mathtt{param}] = \sigma(z) \\
 & & \wedge\ \pi[\sigma(x), \mathtt{meth}] = 1 \wedge H[\sigma(x), \mathtt{meth}] = m
\end{array}$$

Equi-entailment $a \models_{\mathsf{E}} a'$ *holds if:* $\forall \pi, H, \sigma.\ (\pi, H, \sigma \models_{\mathsf{E}} a \implies \pi, H, \sigma \models_{\mathsf{E}} a')$.

Thus, in H', σ' from above we have $\pi, H', \sigma' \not\models_{\mathsf{E}} z.length() = 3$. Furthermore, if (as before) we represent $z.length() \neq 3$ through $(z.length() = 3\,?\,\mathsf{false} : \mathsf{true})$ we obtain $\pi, H', \sigma' \not\models_{\mathsf{E}} z.length() \neq 3$. Note that (since we employ the least fixpoints of the above rules), $\pi, H, \sigma \models_{\mathsf{E}} e.P$ holds, if $\pi, H, \sigma \models_{\mathsf{E}} \mathit{Body(P)}[e/\mathsf{this}]$ holds in a *finite* unfolding. Therefore, we also obtain $\pi, H', \sigma' \not\models_{\mathsf{E}} z.Acyclic$.

Moreover, if we represent $\neg(z.length() = 3)$ through $(z.length() = 3) \to \mathsf{false}$, we obtain that $\pi, H', \sigma' \models_{\mathsf{E}} \neg(z.length() = 3)$. This may seem to be a concern, given that $\pi, H', \sigma' \not\models_{\mathsf{E}} z.length() \neq 3$, when encoded as above. These concerns are eliminated once we add the definition of *framed* expressions and assertions. Essentially, we define a judgement which ensures for a given expression or assertion that a particular state holds enough permissions to access all fields, and that all involved function calls and predicate applications have a finite unrolling. We then define an assertion to be *self-framing* if it can only hold in states in which it is framed. In this way, we guarantee that all assertions are either trivially false, or else their semantics will not include calculation of error values, and thus the intuitive semantics of expressions is restored.[2]

[2] The concept of self-framing assertion was introduced in [21], and described algorithmically in [22]; a more abstract formalisation for assertions not including predicates was developed in [17].

Definition 5 (Framed and Self-Framing Equi-Assertions).
An equi-expression e, or assertion a is equi-framed *in a state consisting of H, π and σ, as the least fixpoint satisfying the following equations:*

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathsf{null}$ $\quad \models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathsf{true}$ $\quad \models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathsf{false}$ $\quad \models^{\pi,H,\sigma}_{\mathsf{frmE}} x$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e.f \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \ \wedge \ \pi(\llbracket e \rrbracket_{H,\sigma}, f) > 0$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e.g(e') \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} e' \wedge \models^{\pi,H,\sigma'}_{\mathsf{frmE}} \mathit{Body}(g))$
where $\sigma' = [\mathsf{this}, \mathsf{X} \mapsto \llbracket e \rrbracket_{H,\sigma}, \llbracket e' \rrbracket_{H,\sigma}]$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e = e' \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} e'$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e_1 ? e_2 : e_3 \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e_1 \ \wedge \ (\pi, H, \sigma \models_{\mathsf{E}} e_1 \Rightarrow \models^{\pi,H,\sigma}_{\mathsf{frmE}} e_2)$
$\wedge \ (\pi, H, \sigma \not\models_{\mathsf{E}} e_1 \ \Rightarrow \models^{\pi,H,\sigma}_{\mathsf{frmE}} e_3)$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathbf{acc}(e.f, q) \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e \to a \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \ \wedge \ (\pi, H, \sigma \models_{\mathsf{E}} e \Rightarrow \models^{\pi,H,\sigma}_{\mathsf{frmE}} a)$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} a_1 * a_2 \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} a_1 \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} a_2$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} e.P \iff \models^{\pi,H,\sigma}_{\mathsf{frmE}} e \wedge \models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathit{Body}(P)[e/\mathsf{this}]$

$\models^{\pi,H,\sigma}_{\mathsf{frmE}} \mathit{Thread}(x, m, y, z) \iff \mathsf{true}$

An equi-expression e is framed by an assertion a, *written* $a \models_{\mathsf{frmE}} e$, *if, (for all π, H, σ) we have $\pi, H, \sigma \models_{\mathsf{E}} a$ implies that $\models^{\Pi,H,\sigma}_{\mathsf{frmE}} e$.*
An equi-assertion a is self-framing, *written* $\models_{\mathsf{frmE}} a$ *if, for all H, π and σ:*
$\pi, H, \sigma \models_{\mathsf{E}} a \ \Rightarrow \ \models^{\pi,H,\sigma}_{\mathsf{frmE}} a$

Thus, $x = 3$ is always framed, and thus self-framing, as is $\mathbf{acc}(x.f, q)$, while $x.next \neq \mathsf{null}$ is framed only when the state holds permission to the heap location $x.next$. Moreover, note that the expressions $x.next = x.next$ and $x.next \neq x.next$ (encoded as earlier), and the assertion $\neg(x.next = x.next)$ are *not* self-framing. Similarly, $x.List$ and $x.length()$ are framed only in states in which we hold the permissions to all the fields in the list, and where x is an acyclic list. In particular, $x.List * x.length() = 4$ is self-framing (note that self-framedness only implies a restriction on the states in which the assertion is true; for a cyclic list structure, the assertion $x.List$ will be false). The assertion $bad() = 4$ is self-framing; intuitively because its semantics does not depend on the heap.

The definitions of this section give a direct semantics to well-defined recursive functions and predicates, in which (terminating) function calls are always equal to their bodies, and predicates are also evaluated in terms of their bodies directly. This semantics is unsuitable for usage in an implementation for three reasons:

1. Treating a predicate instance as meaning its full unrolling yields an unimplementable semantics for checking assertions' truth or entailment; the unbounded unrolling of such a definition cannot be performed in a static tool.
2. Treating a function call as always equal to its body also naturally leads to unbounded instantiation of the recursive definition, in order to evaluate the meaning of assertions in which the function is called.
3. Checking well-definedness and framing of predicate and function definitions is also not practical, since (according to Definition 5), this also depends on being able to evaluate the entire unrolling of the definitions.

In the next section, we turn to the corresponding isorecursive notions, to address these issues.

Discussion. In the remainder of this section, we reflect on some considered design alternatives. In earlier attempts to define a suitable semantics, we considered making the evaluation of expressions (particularly functions) dependent on holding sufficient permissions to *frame* the expressions. This seems intuitive, since reading heap locations without permissions can, at runtime, yield an undefined result (due to race conditions), while evaluating non-terminating functions is clearly undefined at runtime. However, once expression semantics is allowed to be undefined, one either needs to allow the assertion semantics to also provide undefined results, or to provide rules for when to promote undefined results to true or false. The use of additional error values $\mathsf{error}_{\mathsf{tp}}$ allowed us to avoid this; instead taking an "optimistic" semantics of expressions, and then enforcing framedness separately.

Verification tools often require abstraction functions to have a *precondition*, which is a predicate which guarantees that the function body terminates, and that the context of a call will hold sufficient permissions. This is natural for the modelling of partial functions, however, we found that our equirecursive semantics could exploit the use of error values (which implicitly make all functions total) to avoid explicit preconditions. This is consistent with an optimistic expression semantics, and it is not the semantics of this section which needs to be readily checkable in static tools. Preconditions for functions will appear in the isorecursive semantics of the next section.

3 Isorecursive Semantics for Predicates and Functions

In this section, we introduce an assertion semantics in the iso-recursive style. In the iso-recursive approach, predicates are differentiated from their bodies; this is handled in the logic by treating predicate names merely as another kind of permission, which can be rewritten into the corresponding body of the predicate by explicit extra *fold* and *unfold* statements. There are then two different notions of permission a thread can have; the *explicit* permissions, which can be represented in a permission mask as usual, and the *implicit* permissions, which are those which are folded (perhaps recursively) inside predicate instances to which explicit permission is held. Moreover, in the iso-world, functions are equipped with preconditions, which control when the function may be called; the precondition ensures (when it holds) that the body of the function is well-defined.

Running Example for the Isorecursive Setting. In the isorecursive setting the predicate *List*, and the functions *length* and *bad* are defined as follows:

$$
\begin{array}{lll}
\mathit{Body(List)} & \equiv \mathbf{acc}(\mathsf{this}.\mathit{next}, 1)* & \\
 & \quad (\mathsf{this}.\mathit{next} = \mathsf{null}\,?\,\mathsf{true} : \mathbf{acc}(\mathsf{this}.\mathit{next}.\mathit{List})) & \\
\mathit{Pre(length)} & \equiv \mathbf{acc}(\mathsf{this}.\mathit{List}) & \\
\mathit{Body(length)} & \equiv \mathbf{unfolding}\,\mathsf{this}.\mathit{List}\,\mathbf{in} & \\
 & \quad (\mathsf{this}.\mathit{next} = \mathsf{null}\,?\,0 : 1 + \mathsf{this}.\mathit{next}.\mathit{length}()) & \\
\mathit{Pre(bad)} & \equiv \mathsf{true} \qquad\qquad \mathit{Body(bad)} \equiv 1 + \mathit{bad}() &
\end{array}
$$

To differentiate iso-recursive definitions from their equirecursive counterparts, we typically use corresponding upper-case metavariables (e.g., E for expressions).

Definition 6 (Iso-Expressions and Iso-Assertions). *We define the syntax of* iso-expressions *(ranged over by E) and* iso-assertions *(ranged over by A), by the following grammars:*

$$\begin{aligned} E &::= \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid x \mid E.f \mid E.g(E) \mid E = E \mid (E\,?\,E : E) \\ &\quad \mid \mathbf{unfolding}\, E.P\, \mathbf{in}\, E \\ A &::= E \mid \mathbf{acc}(E.f, q) \mid E \to A \mid A * A \mid \mathbf{acc}(E.P) \mid \mathit{Thread}(x, m, y, z) \end{aligned}$$

The additional lookup function Pre(g) retrieves the precondition *of function g.*

The syntax of iso-expressions matches that of equi-expressions, with the exception of $\mathbf{unfolding}\, E_1.P\, \mathbf{in}\, E_2$, which is new here. The value of this expression is the same as that of E_2, however, the way such an expression is checked to be *framed* in a state is different; the unfolding of the predicate P means that E_2 may depend on heap locations whose permissions come from the body of $E_1.P$.

The difference between equi-assertions and iso-assertions is the replacement of the assertion of predicate $e.P$, by *permissions* to such predicate instances, denoted $\mathbf{acc}(E.P)$ above. The semantics of $e.P$ differs from that of $\mathbf{acc}(E.P)$, in that the former unfolds all recursive definitions, while the latter only requires permission to the predicate instance, as will be shown in Definition 9. To keep the presentation simple, we only support *full* permissions to predicates; *i.e.* we do not allow predicate instances themselves to be "split". Some tools support this, and the corresponding extension of our model would be straightforward.

Definition 7 (Semantics of Iso-Expressions). *We define the evaluation of iso-expressions E in a state comprising of heaps H, and environment σ, in the analogous manner to Definition 3, for example,*

$$E = E' \Downarrow_{H,\sigma} \mathsf{false} \quad \text{if } E \Downarrow_{H,\sigma} v \text{ and } E \Downarrow_{H,\sigma} v' \text{ and } v \neq v'.$$

along with the following additional case:

$$\mathbf{unfolding}\, E_1.P\, \mathbf{in}\, E_2 \Downarrow_{H,\sigma} v \quad \text{if } E_2 \Downarrow_{H,\sigma} v.$$

Moreover, as in Definition 3, if E has type tp*, then*

$$[\![E]\!]_{H,\sigma} = v, \quad \text{if } E \Downarrow_{H,\sigma} v, \quad \text{and } \mathsf{error}_{\mathsf{tp}}, \text{ if no such } v \text{ exists.}$$

The full definition appears in the technical report [23].

Thus, in a state H_1, σ_1 in which z points to an acyclic list of two elements, we would have $[\![z.length()]\!]_{H_1,\sigma_1}$=2, and in H', σ' from the discussion following Definition 4, we would have $[\![z.length()]\!]_{H',\sigma'} = \mathsf{error}_{\mathsf{int}}$.

In order to model iso-assertions, we extend the concept of permission mask, so that it also holds permissions to predicate instances.

Definition 8 (Iso-Permissions and Permissions Collection). Isorecursive permission masks, *$\Pi \in Perms$, are mappings from pairs of object or thread identifiers and field names to non-negative values in $\mathbb{Q}$, and from pairs of object identifiers and predicate identifiers to non-negative values in $\mathbb{Z}$.*

The function $\mathcal{P}_{\mathsf{I}}$ *collects the permissions* explicitly *required by an iso-assertion,*

$$\mathcal{P}_{\mathsf{I}} \;:\; \mathit{IsoAssertion} \times \mathit{Heap} \times \mathit{Env} \to \mathit{Perms}$$

$$\begin{array}{ll}
\mathcal{P}_{\mathsf{I}}(E, H, \sigma) & = \emptyset \\
\mathcal{P}_{\mathsf{I}}(\mathbf{acc}(E.f, q), H, \sigma) & = \{\ (\llbracket E \rrbracket_{H,\sigma}, f) \mapsto q\ \} \\
\mathcal{P}_{\mathsf{I}}(E \to A, H, \sigma) & = \mathcal{P}_{\mathsf{I}}(A, H, \sigma) \ \textit{if}\ \llbracket E \rrbracket_{H,\sigma} = \mathit{true}, \quad \emptyset\ \textit{otherwise} \\
\mathcal{P}_{\mathsf{I}}(A * A', H, \sigma) & = \mathcal{P}_{\mathsf{I}}(A, H, \sigma) + \mathcal{P}_{\mathsf{I}}(A', H, \sigma) \\
\mathcal{P}_{\mathsf{I}}(\mathbf{acc}(E.P), H, \sigma) & = \{\ (\llbracket E \rrbracket_{H,\sigma}, P) \mapsto 1\ \} \\
\mathcal{P}_{\mathsf{I}}(\mathit{Thread}(x, m, y, z), H, \sigma) & = \{\ (\sigma(x), \mathtt{recv}) \mapsto 1, (\sigma(x), \mathtt{param}) \mapsto 1, \\
 & \qquad (\sigma(x), \mathtt{meth}) \mapsto 1\}
\end{array}$$

The operation $\odot$*, applied to permission mask* Π*, address* ι *and predicate identifier* P *in a heap* H*, is defined only when* $\Pi(\iota, P) \geq 1$*; it removes the permission to* $\iota.P$*, and adds all the permissions obtained by unfolding the predicate body once:*

$$\Pi \odot_H \iota.P = \Pi[(\iota, P) \mapsto \Pi(\iota, P) - 1] + \mathcal{P}_{\mathsf{I}}(\mathit{Body}(P), H, \sigma), \quad \textit{where}\ \ \sigma(\mathsf{this}) = \iota.$$

As for the equirecursive case, we treat permission masks Π liberally, allowing permissions to fields to be *any* rational numbers, even if they exceed 1. Because we do not work with full knowledge of the permissions contained within predicate instances, in general we cannot rule out the possibility that assertions implicitly require more than the full permission to a field; there is always the possibility for isorecursive permission masks to have no corresponding well-formed equirecursive permission mask. For uniformity, therefore, we ignore this issue in our isorecursive model, and address it in the following section.

Definition 9 (Semantics of Iso-Assertions). *We define the semantics of iso-assertions in a state comprising of an iso-permission mask* Π*, heap* H*, and an environment* σ*, as the smallest fixed point satisfying the following properties:*

$$\begin{array}{lll}
\Pi, H, \sigma \models_{\mathsf{I}} E & \iff & \llbracket E \rrbracket_{H,\sigma} = \mathsf{true} \\
\Pi, H, \sigma \models_{\mathsf{I}} \mathbf{acc}(E.f, q) & \iff & \Pi(\llbracket E \rrbracket_{H,\sigma}, f) \geq q \\
\Pi, H, \sigma \models_{\mathsf{I}} E \to A & \iff & \Pi, H, \sigma \models_{\mathsf{I}} E \ \Rightarrow\ \Pi, H, \sigma \models_{\mathsf{I}} A \\
\Pi, H, \sigma \models_{\mathsf{I}} A_1 * A_2 & \iff & \exists \Pi_1, \Pi_2 : \Pi = \Pi_1 + \Pi_2 \ \wedge\ \Pi_1, H, \sigma \models_{\mathsf{I}} A_1 \\
 & & \qquad \wedge\ \ \Pi_2, H, \sigma \models_{\mathsf{I}} A_2 \\
\Pi, H, \sigma \models_{\mathsf{I}} \mathbf{acc}(E.P) & \iff & \Pi(\llbracket E \rrbracket_{H,\sigma}, P) \geq 1 \\
\Pi, H, \sigma \models_{\mathsf{I}} \mathit{Thread}(x, m, y, z) & \iff & \Pi[\sigma(x), \mathtt{recv}] = 1 = \Pi[\sigma(x), \mathtt{param}] \\
 & & \qquad \wedge\ \ H[\sigma(x), \mathtt{recv}] = \sigma(y) \\
 & & \qquad \wedge\ \ H[\sigma(x), \mathtt{param}] = \sigma(z) \\
 & & \qquad \wedge\ \ \Pi[\sigma(x), \mathtt{meth}] = 1 \wedge H[\sigma(x).\mathtt{meth}] = m
\end{array}$$

Iso-entailment $A \models_{\mathsf{I}} A'$ *holds if:* $\forall \Pi, H, \sigma.\ (\Pi, H, \sigma \models_{\mathsf{I}} A \implies \Pi, H, \sigma \models_{\mathsf{I}} A')$.

Crucially, the semantics of predicate permissions $\mathbf{acc}(E.P)$ does not involve recursion; it is sufficient to simply check that permission to the *predicate* instance is in the direct permissions. This does not directly enforce that the body of the predicate holds in the current state, but, as we shall show in the next section, we push this concern to the definition of a "good" isorecursive state; since a verifier cannot in general enforce that a recursive definition holds, this has to be pushed to the soundness of the underlying methodology (i.e., the equirecursive semantics).

We can now define the notion of framing for iso-expressions and iso-assertions (cf. Definition 5 for the equi-world).

Definition 10 (Framed and Self-Framing Iso-Assertions).

An iso-expression E, or assertion A is iso-framed *in a state consisting of H, Π and σ, as defined by judgements $\models^{\Pi,H,\sigma}_{\mathsf{frml}} E$ and $\models^{\Pi,H,\sigma}_{\mathsf{frml}} A$. Full definitions are provided in the technical report [23], but all cases of these judgements are analogous to those of Definition 5, with the following exceptions:*

$$\models^{\Pi,H,\sigma}_{\mathsf{frml}} E.g(E') \iff \models^{\Pi,H,\sigma}_{\mathsf{frml}} E \wedge \models^{\Pi,H,\sigma}_{\mathsf{frml}} E' \wedge \Pi, H, \sigma' \models_{\mathsf{I}} Pre(g)$$

where $\sigma' = [(\mathsf{this} \mapsto [\![E]\!]_{H,\sigma}), (\mathsf{X} \mapsto [\![E']\!]_{H,\sigma})]$

$$\models^{\Pi,H,\sigma}_{\mathsf{frml}} \mathbf{unfolding}\, E.P\, \mathbf{in}\, E' \iff \models^{\Pi,H,\sigma}_{\mathsf{frml}} E \wedge \Pi([\![E]\!]_{H,\sigma}, P) \geq 1 \wedge \models^{\Pi',H,\sigma}_{\mathsf{frml}} E'$$

where $\Pi' = \Pi \odot_H [\![E]\!]_{H,\sigma}.P$

$$\models^{\Pi,H,\sigma}_{\mathsf{frml}} \mathbf{acc}(E.P) \iff \models^{\Pi,H,\sigma}_{\mathsf{frml}} E$$

An iso-expression E is framed by an assertion A, *written $A \models_{\mathsf{frml}} E$, if, (for all Π, H, σ) we have $\Pi, H, \sigma \models_{\mathsf{I}} A$ implies that $\models^{\Pi,H,\sigma}_{\mathsf{frml}} E$.*

An iso-assertion A is self-framing*, written $\models_{\mathsf{frml}} A$ if, for all H, Π and σ:*

$$\Pi, H, \sigma \models_{\mathsf{I}} A \;\Rightarrow\; \models^{\Pi,H,\sigma}_{\mathsf{frml}} A$$

For example, *Thread(x,m,y,z)* and $\mathsf{this}.List$ are self-framing assertions, while $\mathsf{this}.next{=}\mathsf{null}$ is not.

The rule at the heart of the iso-expressions is the one describing framing for $\mathbf{unfolding}\, E.P\, \mathbf{in}\, E'$: it requires that the context holds permission to the predicate $E.P$, and *adds* the permissions from the body of $E.P$ into the currently held permissions, in order to check framedness of E'. Therefore, in a context where $\Pi(\sigma(\mathsf{this}), next) = 0$, and $\Pi(\sigma(\mathsf{this}), List) = 1$, the expression $\mathsf{this}.next$ is not framed, whereas the expression $\mathbf{unfolding}\, \mathsf{this}.List\, \mathbf{in}\, \mathsf{this}.next$ is framed.

Most importantly, the notion of framing no longer requires a recursive traversal of predicate definitions, as opposed to that from definition 5, which required potentially infinite unrolling. This can only be justified with two further ingredients; firstly, that predicate definitions are always self-framing and functions are only applied in contexts where their bodies are guaranteed to terminate, and secondly that holding a predicate always implicitly guarantees that its body holds in the same state. The former of these two ingredients is provided by the following definition, while the second is provided by the notion of "good state" in the next section.

Definition 11 (Well-formed Definitions).

The definition of an iso-predicate P is well-formed*, if $\models_{\mathsf{frml}} Body(P)$.*

The definition of an iso-function g is well-formed*, if:* (1) $\models_{\mathsf{frml}} Pre(g)$*, and* (2) $\forall \Pi, H, \sigma.(\Pi, H, \sigma \models_{\mathsf{I}} Pre(g) \implies (\models^{\Pi,H,\sigma}_{\mathsf{frml}} Body(g) \wedge [\![Body(g)]\!]_{H,\sigma} \neq \mathsf{error}_{\mathsf{tp}}))$*, where* tp *is the return type of g.*

A program is well-formed *if all function and predicate definitions are well-formed.*

Thus, function *bad* is not well-formed; a function with this body could only be well-formed if its precondition were false. Moreover, *length* is well-formed, but it would not be well-formed if its precondition were true.

Discussion. Note that our notion of well-formedness can be checked *without* unrolling definitions recursively: although criterion (2) of Definition 11 requires that $[\![Body(g)]\!]_{H,\sigma} \neq \mathsf{error}_{\mathsf{tp}}$, this criterion need not (and cannot) be *checked* by fully evaluating the function definition. The special value $\mathsf{error}_{\mathsf{tp}}$ is obtained only if the (least fixpoint of the) rules of Definition 7 do not yield a value for the body. Thus, it is sufficient to impose any conservative termination criterion on the definition of functions, in order to guarantee that $\mathsf{error}_{\mathsf{tp}}$ never arises. In the next section, we will show how the isorecursive definitions above correctly approximate the corresponding equirecursive notions.

We considered making the error value, $\mathsf{error}_{\mathsf{tp}}$, an (unknown) element of the type tp. This approach of *underspecification* is taken in some classical logic based handlings of partial functions. This would work correctly for the semantics of assertions, but would not work correctly for well-formedness (Definition 11), since we check that functions do not evaluate to this value. For example, equating $\mathsf{error}_{\mathsf{bool}}$ with true or with false, would turn a boolean function g, with $Body(g) = x > 3$, into a badly-formed function.

Returning to the three points outlined at the end of the previous section, we can see that our isorecursive definitions directly handle the first and third points of the list (the semantics of predicates, and the checking of framing of expressions and assertions), without requiring a recursive unrolling of any definition. These can therefore be implemented effectively in a static tool. With respect to the second point of our list, our isorecursive expression semantics still defines function calls directly in terms of their bodies, which is not yet suitable for a (matching-loop-free) implementation. Tools handle this problem by applying a variety of additional heuristics, ghost code markers, or triggers [12], in order to implement a constrained version of the definition we employ here; the semantics above is approximated in some way. Since a variety of techniques tackle this problem, we decided not to prescribe a particular strategy. Nonetheless, building a practical restriction of this definition is feasible; in particular, it is possible for a reasonably complete handling of this definition to be achieved based only on the folding and unfolding of *predicates* [7].

4 Comparing the Assertion Semantics

We now turn to relating our two semantics. Our eventual goal is to define an *erasure*, mapping our isorecursive constructions to their equirecursive counterparts, in order to show that verification based on isorecursive semantics gives a sound approximation of verification based on equirecursive semantics. In particular, we will show that fold, unfold and unfolding (which are essential for defining semantics in the isorecursive sense) can all be eliminated from the language, and the resulting program still satisfies the erased version of its specifications. Since

permissions (let alone permissions to predicates) are not reflected at runtime, this leads us closer to a runtime model suitable for proving soundness with respect to an operational semantics. In this section, we focus on the relationship between the two semantics for assertions.

Definition 12 (Encoding). *The encoding* $\langle\!\langle \_ \rangle\!\rangle$ *maps isorecursive expressions and assertions to their equirecursive counterparts, typically by injection, e.g.*
$\langle\!\langle x \rangle\!\rangle = x, \quad \langle\!\langle E.f \rangle\!\rangle = \langle\!\langle E \rangle\!\rangle.f, \quad$ *and* $\quad \langle\!\langle \mathbf{acc}(E.f, q) \rangle\!\rangle = \mathbf{acc}(\langle\!\langle E \rangle\!\rangle.f, q).$
For the cases specific to iso-expressions and assertions, we have
$\langle\!\langle \mathbf{unfolding}\, E.P\, \mathbf{in}\, E' \rangle\!\rangle = \langle\!\langle E' \rangle\!\rangle, \quad$ *and* $\quad \langle\!\langle \mathbf{acc}(E.P) \rangle\!\rangle = \langle\!\langle E \rangle\!\rangle.P.$
The full definition appears in the technical report [23].

Unfolding expressions are unnecessary in the equirecursive world, where predicates and their bodies are not differentiated between. However, the unrolling of predicate definitions may still lead to the discovery that too many field permissions were implicitly held in an isorecursive state; not all isorecursive masks have an equirecursive analogue. This is described in the next definition.

Definition 13 (Encoding permissions). *The (partial) translation* $\langle\!\langle \_ \rangle\!\rangle$ *encodes isorecursive permission maps* Π *into equirecursive permission maps* π*,*

$$\langle\!\langle \Pi \rangle\!\rangle_H = \{(\iota, f) \mapsto q \mid \Pi[\iota, f] = q\} + \{(t, f) \mapsto q \mid \Pi[t, f] = q\} + \langle\!\langle \Pi' \rangle\!\rangle_H$$
$$\text{where } \Pi' = \textstyle\sum_{\Pi(\iota.P)>0} (\mathcal{P}_{\mathsf{I}}(Body(P), H, [\mathsf{this} \mapsto \iota]))$$

Note that $\langle\!\langle \Pi \rangle\!\rangle_H$ may not be well-defined (if a predicate instance held in Π has an infinite unfolding; then the definition above will not terminate), and even when defined, it may not yield a well-formed equi-permissions-mask (cf. Definition 2), if too many field permissions are accumulated. Furthermore, it could be that the constraints required in the bodies of predicates are not always correctly reflected in an iso-recursive state. Thus, we define the notion of a "good" isorecursive state.

Definition 14 (Good Iso-States). *An isorecursive state defined by heap* H*, iso-permissions-mask* Π *and environment* σ *is "good", if:*

1. $\langle\!\langle \Pi \rangle\!\rangle_H$ *is defined, and satisfies* $\models (\langle\!\langle \Pi \rangle\!\rangle_H)$.
2. *For all* ι, P *such that* $\Pi[\iota, P] > 0$, $(\langle\!\langle \Pi \rangle\!\rangle_H), H, \sigma' \models_{\mathsf{E}} w.P$ *is satisfied, where* w *is a fresh variable, and* σ' *is the environment* σ *extended with the mapping* $[w \mapsto \iota]$.

Note, in particular, that the second of these two requirements enforces that the original state does not hold permission to any predicate instance whose definition can be unrolled infinitely. As motivated in Section 2, such predicates are interpreted as false in the equirecursive semantics in any case, so ruling out such states in advance is consistent with this view. In general, good iso-states are those which have a meaningful corresponding equirecursive counterpart.

In Lemma 1 we show that we can map a judgement back from the equirecursive world to the isorecursive, starting from a "good" state in the isorecursive world. Then, in Theorem 1, we show our erasure results.

Lemma 1. *In a well-formed program the following properties hold:*

1. *If* $\Pi(\iota.P) > 0$, *then* $\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \langle\!\langle\, \Pi \odot_H \iota.P \,\rangle\!\rangle_H$.
2. *If* $\pi, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A \,\rangle\!\rangle$, *then* $\exists \Pi$ *s.t.* (Π, H, σ) *is a good iso-state,* $\langle\!\langle\, \Pi \,\rangle\!\rangle_H = \pi$, *and* $\Pi, H, \sigma \models_{\mathsf{I}} A$.

Theorem 1 (Erasure Results).

1. $[\![\langle\!\langle\, E \,\rangle\!\rangle]\!]_{H,\sigma} = [\![E]\!]_{H,\sigma}$.
2. *If* $\Pi, H, \sigma \models_{\mathsf{I}} A$ *and* (Π, H, σ) *is a good iso-state, then* $\langle\!\langle\, \Pi \,\rangle\!\rangle_H, H, \sigma \models_{\mathsf{E}} \langle\!\langle\, A \,\rangle\!\rangle$.
3. *If* (Π, H, σ) *is a good iso-state and* $\models^{\Pi,H,\sigma}_{\mathsf{frmI}} A$, *then* $\models^{\langle\!\langle\, \Pi \,\rangle\!\rangle,H,\sigma}_{\mathsf{frmE}} \langle\!\langle\, A \,\rangle\!\rangle$.
4. *If all functions and predicates are well-formed, then if an iso-assertion* A *is self-framing, then* $\langle\!\langle\, A \,\rangle\!\rangle$ *is self-framing.*
5. *If* $A \models_{\mathsf{I}} A'$, *then* $\langle\!\langle\, A \,\rangle\!\rangle \models_{\mathsf{E}} \langle\!\langle\, A' \,\rangle\!\rangle$.

The proof sketches are given in the technical report [23]. These results demonstrate that the more-readily-checkable isorecursive notions from the previous section accurately approximate the intended underlying equirecursive notions. In the following sections, we extend this argument to Hoare Logics for a small language. We will then be in a position to prove that verifying a program with respect to *isorecursive* definitions, is sufficient to guarantee soundness, via our erasure results and a soundness proof with respect to equirecursive semantics.

5 Hoare Logic for Iso-Assertions

In this section, we define an axiomatic semantics for a small (but representative) subset of Chalice, with respect to our iso-recursive assertion semantics (as defined in Section 3). This is the semantics closest to most implementations, but it is not so intuitive or useful as a runtime model. In the following section, we will define a corresponding Hoare Logic for our equirecursive semantics, and demonstrate the relationship between the two.

5.1 Chalice Syntax

We begin by defining our Chalice subset:

Definition 15 (Isorecursive Chalice Syntax). *We assume a set of predefined methods, ranged over by* m. *For simplicity, methods are assumed to have exactly one parameter, and to always return a value. Furthermore, method names are assumed to be unique in the whole program.*
Simple statements*, ranged over by* R*, and* statements*, ranged over by* S*, are mutually defined by the following grammars:*

$$
\begin{array}{rcl}
R & ::= & \texttt{skip} \mid x{:=}E \mid x.f{:=}y \mid \texttt{return}\ x \mid x{:=}\ \texttt{new}\ c \mid (\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \\
 & \mid & x{:=}\ \texttt{fork}\ y.m(z) \mid y{:=}\ \texttt{join}\ x \mid \textbf{fold}\, x.p \mid \textbf{unfold}\, x.p \\
S & ::= & R \mid (R; S)
\end{array}
$$

A statement S is return-ended *if the right-most simple statement occurring in its structure is of the form* `return` x*; i.e., it can be constructed by the following sub-grammar:* $S ::= \texttt{return}\ x \mid (R; S)$
Composition of statements S_1 *and* S_2*, for which we use the (overloaded) notation* $(S_1; S_2)$*, results in a statement which represents appending the two sequences of simple statements; i.e., when* S_1 *is not a simple statement (say,* $S_1 = (R; S')$*), is defined by recursively rewriting* $((R; S'); S_2) = (R; (S'; S_2))$.

Our syntax only allows for sequential compositions to be nested to the right, which simplifies the definition of the operational semantics (see Section 7), since we do not need a separate concept of evaluation contexts for such a simple language. Our language only allows general expressions e within variable assignments, and otherwise employs only program variables for expressions, but the generalisation is easily encoded (or made). Multi-threading is achieved by the ability to *fork* and *join* threads. The statement $w{:=}\ \texttt{fork}\ m.y(z)$ has the meaning of starting an invocation of a call to method m (with receiver y and parameter z) in a new thread. The returned value (stored in w) is a *token*, which gives a means of referring to this newly-spawned thread. Such a token can be used to *join* the thread later (which has the operational meaning of waiting for the thread to terminate, and then receiving its return value); this is provided by the $x{:=}\ \texttt{join}\ w$ statement.

We do not include loops, since they present no relevant challenges compared with recursion. While we do not support a method call statement, we do allow the fork and join of method invocations. A normal method call of the form $x{:=}\ m.y(z)$ can be encoded by a sequence $(w{:=}\ \texttt{fork}\ m.y(z)\ ; x{:=}\ \texttt{join}\ w\ )$ (for some fresh variable w).

5.2 Hoare Logic

We now define the Hoare Logic corresponding to isorecursive assertion semantics.

Definition 16 (Isorecursive Hoare Logic). Isorecursive Hoare Triples *are written* $\vdash_I \{A\}\ S\ \{A'\}$*, where* A *and* A' *are self-framing isorecursive assertions, and* S *is an Isorecursive Chalice statement. The rules are shown in Figure 1. We leave implicit the requirement that* A *and* A' *are always self-framing; in particular, whenever we write a triple (even as a new conclusion of a derivation rule), this requirement must be satisfied in addition to the explicit premises.*

Our Hoare triples employ isorecursive assertions as pre- and post-conditions, with the restriction that the assertions used must always be *self-framing*. The restriction to self-framing assertions is important for soundness. For example, without this requirement, it would naturally be possible to derive triples such as $\vdash_I \{x.f = 1\}\ (\texttt{skip}; \texttt{skip})\ \{x.f = 1\}$, which, when evaluated at runtime, might not be sound (another thread could modify the location $x.f$ during execution). Indeed, in our soundness proof, the requirement that every thread has a self-framing pre-condition is essential to the argument.

$$\frac{}{\vdash_I \{A\}\ \texttt{skip}\ \{A\}}\ (skipI) \qquad \frac{A[E/x] \models_{\mathsf{frml}} E}{\vdash_I \{A[E/x]\}\ x{:=}E\ \{A\}}\ (varassI)$$

$$\frac{}{\vdash_I \{x \neq \mathsf{null} * \mathbf{acc}(x.f,1)\}\ x.f{:=}y\ \{\mathbf{acc}(x.f,1) * x.f = y\}}\ (fldassI)$$

$$\frac{}{\vdash_I \{A\}\ \texttt{return}\ x\ \{A * \mathsf{result} = x\}}\ (retI)$$

$$\frac{\overline{f_i} = fields(c)}{\vdash_I \{\mathsf{true}\}\ x{:=}\ \texttt{new}\ c\ \{x{\neq}\mathsf{null}\ *\ (*\overline{\mathbf{acc}(x.f_i,1)})\}}\ (newI)$$

$$\frac{\vdash_I \{A * B\}\ S_1\ \{A'\} \quad \vdash_I \{A * \neg B\}\ S_2\ \{A'\}}{\vdash_I \{A\}\ (\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2)\ \{A'\}}\ (ifI)$$

$$\frac{A = Pre(m)[y/\mathsf{this}][z/\mathsf{X}]}{\vdash_I \{y \neq \mathsf{null} * A\}\ x{:=}\ \texttt{fork}\ y.m(z)\ \ \{Thread(x,m,y,z)\}}\ (forkI)$$

$$\frac{A' = Post(m)[y/\mathsf{this}][z/\mathsf{X}][w/\mathsf{result}]}{\vdash_I \{Thread(x,m,y,z)\}\ w{:=}\ \texttt{join}\ x\ \ \{A'\}}\ (joinI)$$

$$\frac{\vdash_I \{A\}\ R\ \{A'\} \quad \vdash_I \{A'\}\ S\ \{A''\}}{\vdash_I \{A\}\ (R;S)\ \{A''\}}\ (seqI)$$

$$\frac{A_1 \models_{\mathsf{I}} A_3 \quad \vdash_I \{A_3\}\ S\ \{A_4\} \quad A_4 \models_{\mathsf{I}} A_2}{\vdash_I \{A_1\}\ S\ \{A_2\}}\ (consI)$$

$$\frac{\vdash_I \{A\}\ S\ \{A'\} \quad \models_{\mathsf{frml}} A'' \quad mods(S) \cap FV(A'') = \emptyset}{\vdash_I \{A'' * A\}\ S\ \{A' * A''\}}\ (frameI)$$

$$\frac{\models_{\mathsf{frml}} A \quad A' = Body(P)[x/\mathsf{this}]}{\vdash_I \{A * A' * B\}\ \mathbf{fold}\,\mathbf{acc}(x.P,q)\ \{A * \mathbf{acc}(x.P,q) * \mathbf{unfolding}\,x.P\,\mathbf{in}\,B\}}\ (foldI)$$

$$\frac{\models_{\mathsf{frml}} A \quad A' = Body(P)[x/\mathsf{this}]}{\vdash_I \{A * \mathbf{acc}(x.P,q) * \mathbf{unfolding}\,x.P\,\mathbf{in}\,B\}\ \mathbf{unfold}\,\mathbf{acc}(x.P,q)\ \{A * A' * B\}}\ (unfoldI)$$

Fig. 1. Hoare Logic for Isorecursive semantics

Some of the rules (such as the treatment of conditionals, and the rule of consequence, (*consI)*)) are standard, but others warrant discussion. The *frame rule* (*frameI*) (whose role is to preserve parts of the state which are not relevant for the execution of the particular statement) is similar to that typically employed in separation logics [8]. The extra assertion A'' can be "framed on" under two conditions; firstly, that no variables mentioned in A'' are modified by the statement S, and secondly, that A'' is self-framing. The two conditions are necessary for similar reasons; if the execution of S could change the meaning of A'', then

to simply conjoin it unchanged to both pre- and post-condition would be incorrect. The two ways in which the state can change in our language are through variable assignments (whose effects are tamed by the first requirement) and field assignments (which cannot affect the meaning of A'', since A'' is self-framing, and therefore comes along with sufficient permission to rule out assignment to the fields on which its meaning depends).

The (*varassI*) rule is similar to a standard Hoare Logic rule for assignment, but with the extra requirement that the expression to be assigned is readable in the pre-condition state. The premise guarantees that fields are only read when appropriate permissions are known to be available, and functions are only applied when their pre-conditions are known in the state. The rule (*fldassI*) is slightly subtle: it must avoid the possibility of outdated heap-dependent expressions surviving the assignment; the requirement for *full* permission to the written field location from the pre-condition forces any information previously-known about that location to be discarded (i.e., using rule (*consI*)) prior to applying this rule).

The rules for forking and joining threads employ the special $\mathit{Thread}(x, m, y, z)$ assertion, discussed above. Our formulation does not allow this knowledge to be split amongst threads (though it can be passed from thread to thread in contracts); extensions are possible but orthogonal to the topic of this paper.

The two most important rules for the isorecursive semantics are those for folding and unfolding predicate instances. For example, consider folding a predicate instance, as defined by rule (*foldI*)). It is easy to see that this rule should exchange the body of the predicate instance (the assertion A' for a permission to the predicate itself). The challenge is to enable the preservation of information which was previously framed by the predicate's contents, even though those permissions have (after folding) been stored into the predicate body. For example, consider a predicate P whose definition is $\mathbf{acc}(\mathsf{this}.g, 1)$. In a state in which we know $\mathbf{acc}(\mathsf{this}.g) * \mathsf{this}.g = 4$, we could not treat a fold of P as a simple exchange of $\mathbf{acc}(\mathsf{this}.g)$ for $\mathbf{acc}(\mathsf{this}.P)$, since, in the resulting state, $\mathsf{this}.g = 4$ would not be framed. Instead, our rule allows us to derive the post-condition $\mathbf{acc}(\mathsf{this}.P) * \mathbf{unfolding}\,\mathsf{this}.P\,\mathbf{in}\,\mathsf{this}.g = 4$, which is self-framing. Furthermore, in order to handle the possibility that we wish to preserve an expression which is framed *partially* by the permissions required by a predicate body, we allow the presence of a further assertion A in the rule. This allows, *e.g.* a pre-condition such as $\mathbf{acc}(\mathsf{this}.f) * \mathbf{acc}(\mathsf{this}.g) * \mathsf{this}.f = \mathsf{this}.g$ for a statement $\mathbf{fold}\,\mathbf{acc}(\mathsf{this}.P)$ to be used to derive a post-condition $\mathbf{acc}(\mathsf{this}.f) * \mathbf{acc}(\mathsf{this}.P) * \mathbf{unfolding}\,\mathsf{this}.P\,\mathbf{in}\,\mathsf{this}.f = \mathsf{this}.g$, in which the additional assertion ($\mathbf{acc}(\mathsf{this}.f)$ in this case) provides additional permissions required for self-framing. The condition that A must be self-framing is necessary to avoid that A itself might represent information which was only framed by permissions from the body of P.

The rule for unfolding predicates is exactly symmetrical with that for folding predicates. In particular, it enables information from unfolding expressions

depending on the predicate to be unfolded, to be preserved (without an unfolding expression) in the post-state.

We can characterise the derivable triples in our Hoare Logic, using a *generation lemma*; an example case is shown here.

Lemma 2 (Generation Lemma).

- $\vdash_I \{A\}\ x{:=}E\ \{A'\}\ \Leftrightarrow$
  $\exists A''.(\ \models_{\mathsf{frml}} A''\ \ \wedge\ \ A \models_{\mathsf{I}} A''[E/x]\ \ \wedge\ \ A''[E/x] \models_{\mathsf{frml}} E\ \ \wedge\ \ A'' \models_{\mathsf{I}} A')$
- *Remaining cases in the technical report [23].*

6 Hoare Logic for Equi-Assertions

In this section, we employ a second Hoare Logic based on our equirecursive assertion semantics. Firstly, we define an "erased" form of our statement syntax, in which only equirecursive expressions are used, and no fold and unfold statements may occur.

Definition 17 (Equirecursive Chalice Syntax). Simple runtime statements*, ranged over by* r*, and* runtime statements*, ranged over by* s*, are mutually defined by the following grammars:*

$$\begin{array}{rcl} r & ::= & \texttt{skip} \mid x{:=}e \mid x.f{:=}y \mid \texttt{return}\ x \mid x{:=}\ \texttt{new}\ c \mid (\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2) \\ & \mid & x{:=}\ \texttt{fork}\ y.m(z)\ \mid y{:=}\ \texttt{join}\ x \\ s & ::= & r \mid (r; s) \end{array}$$

The notions of return-ended statements *and* composition of statements *are analogous to those of Definition 15.*

We can now define the equirecursive analogue of Definition 16.

Definition 18 (Equirecursive Hoare Logic). Equirecursive Hoare Triples *are written* $\vdash_E \{a\}\ s\ \{a'\}$*, where* a *and* a' *are self-framing equirecursive assertions, and* s *is an equirecursive Chalice statement. The rules are analogous to those of our Isorecursive Hoare Logic (Definition 16), except that the corresponding equirecursive notions of entailment, self-framing, statements, assertions etc. are used throughout. In addition, there are no rules for* **fold** *and* **unfold** *statements (since these do not occur in equirecursive Chalice). The full rules are given in the technical report [23].*

We now extend our previous erasure results (mapping isorecursive to equirecursive assertions) to also define an erasure on statements. This operation applies erasure to all assertions and expressions, and replaces all **fold**/**unfold** statements with `skip`.

Definition 19 (Encoding iso-statements to equi-statements). *We overload the encoding* $\langle\!\langle\ \_\ \rangle\!\rangle$ *to map isorecursive to equirecursive statements, as*:

$$\begin{array}{c} \langle\!\langle\ x{:=}E\ \rangle\!\rangle = x{:=}\langle\!\langle\ E\ \rangle\!\rangle \quad \langle\!\langle\ (S_1; S_2)\ \rangle\!\rangle = (\langle\!\langle\ S_1\ \rangle\!\rangle; \langle\!\langle\ S_2\ \rangle\!\rangle) \\ \langle\!\langle\ (\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2)\ \rangle\!\rangle = (\texttt{if}\ \langle\!\langle\ E\ \rangle\!\rangle\ \texttt{then}\ \langle\!\langle\ S_1\ \rangle\!\rangle\ \texttt{else}\ \langle\!\langle\ S_2\ \rangle\!\rangle) \\ \langle\!\langle\ \textbf{fold}\, x.P\ \rangle\!\rangle = \texttt{skip} = \langle\!\langle\ \textbf{unfold}\, x.P\ \rangle\!\rangle \\ \langle\!\langle\ S\ \rangle\!\rangle = S \quad \textit{otherwise} \end{array}$$

Theorem 2. *If A, A' are self-framing iso-assertions, and S is an isorecursive Chalice statement, then*

$$\vdash_I \{A\}\ S\ \{A'\}\ \Rightarrow \vdash_E \{\langle\!\langle A \rangle\!\rangle\}\ \langle\!\langle S \rangle\!\rangle\ \{\langle\!\langle A' \rangle\!\rangle\}$$

7 Operational Semantics and Soundness

In this section, we show soundness of our formalisations, with respect to an interleaving small-step operational semantics. We formalise our runtime model with respect to a collection of *threads* and *objects*, together referred to as *runtime entities*. We do not model explicit object allocation; instead, we assume that all objects are pre-existing (and already have classes), but have a flag which indicates whether they are truly allocated or not. When unallocated, an object holds the permission to all of its own fields. Thus, we never need to create or destroy permission in the system; it is merely transferred from entity to entity. Similarly, we do not model creation of new threads, but just assume that idle thread entities exist in the system, which can be assigned a task (i.e., a method invocation) to begin executing.

Definition 20 (Runtime Ingredients). *Remember that object identifiers are ranged over by ι, and the (disjoint) set of thread identifiers is ranged over by t. We assume a fixed mapping $cls(\iota)$ from object identifiers to class names.*
A runtime heap *h is a mapping from pairs of object identifier and field name, to values.*
A thread configuration *T is defined by $T ::= \{\sigma, s\} \mid \texttt{idle}$ where s is a return-ended runtime statement (cf. Definition 17).*
A thread entity *is a thread configuration labelled with a thread identifier, T_t. A thread entity is called* active *if it is of the form $\{\sigma, s\}_t$.*
An object state *O is defined by $O ::= \texttt{alloc} \mid \texttt{free}$, and an* object entity *is an object state labelled with an object identifier, written O_ι.*
A labelled entity *N_n is defined by $N_n ::= T_t \mid O_\iota$, where the label n denotes the thread or object identifier of the entity, respectively.*

Note that, in contrast to the heaps of Definition 2, runtime heaps do not store ghost information about thread identifiers. At runtime, this information is directly available via the thread configurations present.

We define two main types of small-step transitions, which we call *local* and *paired* transitions. A local transition is one which affects only a single (thread) entity and the heap.

Definition 21 (Local transitions). Local transitions *map a heap and thread entity to a new heap and thread entity (with the same thread identifier), and are written $h, T_t \xrightarrow{l} h', T'_t$. These rules have the expected shape, e.g.*

$$\frac{\llbracket e \rrbracket_{h,\sigma} = v}{h, \{\sigma, (x{:=}e; s)\}_t \xrightarrow{l} h, \{\sigma[x \mapsto v], s\}_t}\ (varassS)$$

Full rules for local transitions are given in the technical report [23].

The more complex transitions are concerned with forking and joining threads, and with object allocation. In the case of forking and joining, we define transitions which simultaneously involve *two* thread entities; one which is executing the `fork`/`join` statement, and one which represents the thread being forked/joined. In the case of a fork, the second thread entity must be initially idle, while in the case of a join, it must have finished executing and be ready to return. Object allocation, on the other hand, is a transition involving a thread entity and an object entity together; it takes an object entity in the `free` state (and of the appropriate class), and switches it to `alloc`.

Definition 22 (Paired transitions). Paired transitions *map a heap and a pair of entities to a new heap and pair of entities (with the same identifiers), and are written* $h, (T_t \| N_n) \xrightarrow[p]{} h', (T'_t \| N'_n)$.

$$\frac{\sigma(y) = \iota \quad \sigma' = [\mathsf{this} \mapsto \iota, \mathsf{X} \mapsto \sigma(z), \mathsf{method} \mapsto m] \quad s' = \langle\!\langle\, Body(m) \,\rangle\!\rangle}{\begin{array}{c} h, (\{\sigma, (x{:=}\ \texttt{fork}\ y.m(z)\ ; s)\}_{t_1} \| \texttt{idle}_{t_2}) \\ \xrightarrow[p]{} h, (\{\sigma[x{\mapsto}t_2], s\}_{t_1} \| \{\sigma', s'\}_{t_2}) \end{array}}\ (\mathit{forkS})$$

$$\frac{\sigma_1(y) = t_2}{\begin{array}{c} h, (\{\sigma_1, (x{:=}\ \texttt{join}\ y\ ; s)\}_{t_1} \| \{\sigma_2, \texttt{return}\ z\}_{t_2}) \\ \xrightarrow[p]{} h, (\{\sigma_1[x{\mapsto}\sigma_2(z)], s\}_{t_1} \| \texttt{idle}_{t_2}) \end{array}}\ (\mathit{joinS})$$

$$\frac{\mathit{cls}(\iota) = c \quad \overline{f_i} = \mathit{fields}(c) \quad h' = h[\overline{(\iota, f_i) \mapsto \mathsf{null}}]}{h, (\{\sigma, (x{:=}\ \texttt{new}\ c; s)\}_t \| \texttt{free}(c)_\iota) \xrightarrow[p]{} h', (\{\sigma[x{\mapsto}\iota], s\}_t \| \texttt{alloc}(c)_\iota)}\ (\mathit{newS})$$

We can now define the operational semantics for a whole system.

Definition 23 (Runtime Configurations and Operational Semantics). *A* runtime entity collection C *is a pair* $(\overline{T_t}, \overline{O_\iota})$ *consisting of a set of thread entities (one for each thread identifier* t*) and a set of object entities (one for each object identifier* ι*). A* runtime configuration *is a pair* h, C *of a runtime heap and a runtime entity collection.*

The interleaving operational semantics of such a configuration is given by transitive closure of the transition relation $h, C \xrightarrow[c]{} h', C'$*, defined as follows:*

$$\frac{C[t] = T_t \quad h, T_t \xrightarrow[l]{} h', T'_t}{h, C \xrightarrow[c]{} h', C[t{\mapsto}T'_t]}\ (\mathit{selectLocalS})$$

$$\frac{C[t] = T_t \quad C[n] = N_n \quad h, (T_t \| N_n) \xrightarrow[p]{} h', (T'_t \| N'_n)}{h, C \xrightarrow[c]{} h', C[t{\mapsto}T'_t][n{\mapsto}N'_n]}\ (\mathit{selectPairS})$$

In order to reuse our equirecursive assertion logic semantics for runtime configurations, we define a mapping *Heap* back from runtime configurations to heaps as in Definition 2. $\mathit{Heap}(h, C)$ reconstructs the ghost information about threads, from the information in the runtime entity collection, adding it to the heap

information in h. We also require an equirecursive permission collection function $\mathcal{P}_{\mathsf{E}}$, which collects all of the permissions explicitly or implicitly required in equi-assertions a. Both operators are defined in the technical report [23].

We can now turn to the central notion of our soundness proof; what it means for a runtime configuration to be *valid*. Essentially, this prescribes that the permissions to the fields of all allocated objects can be notionally divided amongst the active threads (point 3 below), and suitable preconditions for the statements of each thread can be chosen that are satisfied in the current runtime configuration, and for which each statement can be verified (via our equirecursive Hoare Logic) with respect to the thread's current postcondition.

Definition 24 (Valid configuration). *A runtime configuration* (h, C)*, is* valid *if there exists a set of equirecursive assertions* $\overline{a_t}$*, (one for each thread identifier* t*), such that:*

1. *For each thread entity of the form* $\mathtt{idle}_t$*,* $a_t = \mathsf{true}$*.*
2. *For each thread entity of the form* $\{\sigma, s\}_t$ *in* C*, letting* $H = \mathit{Heap}(h, C)$*, we have both* $\mathcal{P}_{\mathsf{E}}(a_t, H, \sigma), H, \sigma \models_{\mathsf{E}} a_t$ *and* $\vdash_E \{a_t\}\ s\ \{\mathit{Post}(\sigma(\mathsf{method}))\}$*.*
3. $\models (\sum_{\{\sigma,s\}_t \in C} \mathcal{P}_{\mathsf{E}}(a_t, h, \sigma)) + (\sum_{\mathtt{free}(c)_\iota \in C} \sum_{f \in \mathit{fields}(c)} \{(\iota, f) \mapsto 1\}) + (\sum_{\mathtt{idle}_t \in C} \{(t, \mathtt{recv}) \mapsto 1, (t, \mathtt{param}) \mapsto 1, (t, \mathtt{meth}) \mapsto 1\})$

Finally, we can turn to our main soundness result, which shows that modular verification of each definition in the program, using our isorecursive semantics, is sound with respect to the interleaving operational semantics of the language.

Theorem 3 (Soundness of Isorecursive Hoare Logic). *For a well-formed program, if all method definitions satisfy that both* $\vdash_I \{\mathit{Pre}(m)\}\ \mathit{Body}(m)\ \{\mathit{Post}(m)\}$ *and* $\mathit{Body}(m)$ *is a return-ended statement, and if* h, C *is a valid configuration, and if* $h, C \xrightarrow[c]{} h', C'$*, then* h', C' *is a valid configuration.*

Note that the use of our *isorecursive* Hoare Logic here, reflects the fact that a program must be verifiable statically. However, our earlier results easily allow us to connect (in the proof) with the equirecursive notions, which are closer to the actual runtime. A proof sketch is provided in the technical report [23].

8 Related Work and Conclusions

This paper has explored the many challenges involved in handling flexible recursive specification constructs in ways which are both amenable for formal mathematical proofs (the equirecursive setting), and implementation in practical static tools (the isorecursive setting). Our work is set in the context of implicit dynamic frames, which supports an interesting combination of recursive predicates, functions and unfolding expressions, each of which provides additional challenges. However, the issues we have described show up in many other settings, including those which do not support all three features simultaneously.

The first formally-rigorous treatment of recursive predicates in the context of permission-based logics was proposed in [15] for separation logic [8,13]; this treatment was further developed in [16]. In both works, and in many subsequent formal papers, the (only) meaning of recursive predicates is given by the least fix-point of the unrolling of their bodies, i.e., the equirecursive treatment.

Many existing verification tools based on separation logic, such as jStar [5] and VeriFast [9], support custom recursive definitions in the form of abstract predicates. jStar applies a sequence of inbuilt heuristics (which can be user-defined) to decide on the points in the code at which to fold or unfold recursive definition, while VeriFast requires the user to provide **fold** and **unfold** statements explicitly (which can nonetheless be inferred in some cases). The full unrolling of a recursive definition is not made available to the verifier; the isorecursive interpretation is used for the implementations.

The problem of handling partial functions in a setting with only total functions has received much prior attention (see [20] for an excellent summary). We aimed to avoid allowing "undefined" as a possible outcome in our semantics, as explained in Section 2. As an alternative, we could have considered taking the approach of *semi-classical logics* (e.g., [24]), and allowing undefined expressions but not assertions. In a sense, our solution is somewhat similar, since we use the extra $\mathsf{error}_{\mathsf{tp}}$ values to circumvent potential undefinedness for expressions.

The combination of fractional permissions [4] with separation logic for concurrent programming was proposed in [3]. These ideas were adapted to concurrent object oriented programming and formalised in [6], and further adapted to the implicit dynamic frames [21] setting and implemented in the form of Chalice [10]. The Chalice approach has been formalised [19] through a Hoare Logic for implicit dynamic frames. However, neither [6], nor [19] give a treatment of recursive predicates and functions. A verification condition generation semantics for implicit dynamic frames was developed and proven sound in [22].

As future work, we plan to prove soundness of verification tools/methodologies based on the formalisms provided here. We would also like to explore how to connect the notions of isorecursive definitions provided here with other related areas, such as tools for shape and static analysis, in which different but related issues regarding the bounding of recursive definitions arise.

Acknowledgements. We are very grateful to Peter Müller for extensive feedback on an early version of this work, and to the anonymous reviewers for detailed suggestions for improvements to the details and presentation of our formalisms.

References

1. Abadi, M., Fiore, M.P.: Syntactic considerations on recursive types. In: Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS 1996), pp. 242–252. IEEE Computer Society Press (July 1996)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)

3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270 (2005)
4. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
5. DiStefano, D., Parkinson, M.J.: jStar: Towards practical verification for Java. In: OOPSLA. ACM Press (2008)
6. Haack, C., Hurlin, C.: Separation logic contracts for a Java-like language with fork/Join. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 199–215. Springer, Heidelberg (2008)
7. Heule, S., Kassios, I.T., Müller, P., Summers, A.J.: Verification condition generation for permission logics with abstract predicates and abstraction functions. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 451–476. Springer, Heidelberg (2013)
8. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL, pp. 14–26. ACM Press (2001)
9. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
10. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
11. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
12. Moskał, M.: Programming with triggers. In: SMT 2009: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (2009)
13. O'Hearn, P.W., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
14. Parkinson, M.J.: Local Reasoning for Java. PhD thesis, University of Cambridge (November 2005)
15. Parkinson, M.J., Bierman, G.: Separation logic and abstraction. In: POPL, pp. 247–258. ACM Press (2005)
16. Parkinson, M.J., Bierman, G.: Separation logic, abstraction and inheritance. In: POPL, pp. 75–86. ACM Press (2008)
17. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 439–458. Springer, Heidelberg (2011)
18. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. Logical Methods in Computer Science 8(3:01), 1–54 (2012)
19. Raad, A., Drossopoulou, S.: A sip of the chalice. In: FTfJP (July 2011)
20. Schmalz, M.: Formalizing the logic of event-B. PhD thesis, ETH Zurich (November 2012)
21. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
22. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames. In: ToPLAS (2012)
23. Summers, A.J., Drossopoulou, S.: A formal semantics for isorecursive and equirecursive state abstractions. Technical Report 773, ETH Zurich (2012)
24. Farmer, W.M.: A partial functions version of Church's simple theory of types. Journal of Symbolic Logic 55, 1269–1291 (1990)

Trustworthy Proxies
Virtualizing Objects with Invariants

Tom Van Cutsem[1,⋆] and Mark S. Miller[2]

[1] Vrije Universiteit Brussel, Belgium
[2] Google Research, USA

Abstract. Proxies are a common technique to virtualize objects in object-oriented languages. A proxy is a placeholder object that emulates or wraps another target object. Both the proxy's representation and behavior may differ substantially from that of its target object.

In many OO languages, objects may have language-enforced invariants associated with them. For instance, an object may declare immutable fields, which are guaranteed to point to the same value throughout the execution of the program. Clients of an object can blindly rely on these invariants, as they are enforced by the language.

In a language with both proxies and objects with invariants, these features interact. Can a proxy emulate or replace a target object purporting to uphold such invariants? If yes, does the client of the proxy need to trust the proxy to uphold these invariants, or are they still enforced by the language? This paper sheds light on these questions in the context of a Javascript-like language, and describes the design of a Proxy API that allows proxies to emulate objects with invariants, yet have these invariants continue to be language-enforced. This design forms the basis of proxies in ECMAScript 6.

Keywords: Proxies, Javascript, reflection, language invariants, membranes.

1 Introduction

Proxies are a versatile and common abstraction in object-oriented languages and have a wide array of use cases [1]. Proxies effectively "virtualize" the interface of an object (usually by intercepting all messages sent to the object). One common use case is for a proxy to *wrap* another target object. The proxy mostly behaves identical to its target, but augments the target's behavior. Access control wrappers, profilers, taint tracking [2] and higher-order contracts [3] are examples. Another common use case is for a proxy to *represent* an object that is not (yet) available in the same address space. Examples include lazy initialization of objects, remote object references, futures [4] and mock-up objects in unit tests.

Objects in object-oriented languages often have *invariants* associated with them, i.e. properties that are guaranteed to hold for the entire lifetime of the object. Some of these invariants may be enforced by the programming language itself. We will call

⋆ Postdoctoral Fellow of the Research Foundation, Flanders (FWO).

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 154–178, 2013.

these *language invariants*. Examples of such language invariants include immutable object fields, which are guaranteed to point to the same value throughout the object's lifetime; or, in a prototype-based language, an immutable "prototype" link pointing to an object from which to inherit other properties. Clients of an object can blindly rely on these invariants, as they are enforced by the language (that is: the language provides no mechanism by which the invariants can ever be broken). Such invariants are important for developers to reason about code, critical for security, and useful for compilers and virtual machines.

In a language with both proxies and objects with language invariants, these features interact. Can a proxy virtualize an object with language invariants? Can it wrap a target object with language invariants and claim to uphold these invariants itself? If so, does the client of the proxy need to trust the proxy to uphold these invariants, or are they still enforced by the language? This paper sheds light on these questions in the context of Javascript, a language that features both proxies and objects with language invariants.

Contribution. We study the apparent tradeoff between the needs of a powerful interposition mechanism that may break language invariants versus the desire to maintain these invariants. The key contribution of this paper is the design of a Proxy API that allows proxies to virtualize objects with invariants, without giving up on the integrity of these invariants (i.e. they continue to be enforced by the language). To emphasize this, we call these proxies *trustworthy*. We call out the general mechanism of our API, named *invariant enforcement*, that is responsible for this trustworthiness.

Paper Outline. In Section 2, we illustrate how the issues highlighted above came up in the design of a Proxy API for Javascript. We go on to describe the general principle by which our Proxy API enforces invariants. To study this mechanism in detail, we introduce λ_{TP}, an extension of the lambda calculus featuring proxies, in Section 3. We then employ proxies to build various access control abstractions in Section 4. This will allow us to discuss the advantages and drawbacks of our API (Section 5). We end with an overview of the implementation status (Section 6) and related work (Section 7).

2 Trustworthy Proxies: The Case of Javascript

The questions addressed in this paper initially arose while the authors were designing a Proxy API for Javascript[1]. We first highlight the problems we encountered in combining proxies with Javascript's language invariants, and subsequently describe how the Javascript Proxy API deals with these issues.

2.1 Language Invariants in Javascript

Javascript is known as a dynamic language, primarily because its core abstraction – objects with prototype-based inheritance – is extremely flexible. Consider the following object definition:

```
var point = { x: 0, y: 0 };
```

[1] To be precise: for ECMAScript 6, the next standard version of Javascript.

This defines a `point` object with `x` and `y` properties. Javascript objects have very weak language invariants: new properties can be added to existing objects at any time (`point.z = 0;`), existing properties can be updated by external clients (`point.x = 1;`), or even deleted (`delete point.x;`). This makes it challenging for developers to reason about the structure of objects: it is always possible that third-party client objects will modify the object at run-time.

ECMAScript 5 (ES5) [5] added new primitives that allow one to strengthen the invariants of objects, making it possible to more easily reason about the structure of objects, by protecting them from unintended modifications. Figure 1 depicts a simplified state diagram of an ES5 object.

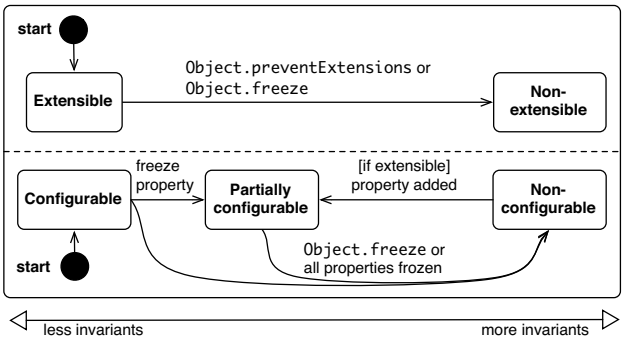


Fig. 1. Simplified state diagram of an ES5 object

ECMAScript 5 defines two independent language invariants on objects:

Non-extensibility. Every ES5 object is born in an *extensible* state. In this state, new properties can be freely added to the object. One can make an object `obj` non-extensible by calling `Object.preventExtensions(obj)`. Once an object is made non-extensible, it is no longer possible to further add new properties to the object. Once an object is non-extensible, it remains forever non-extensible.

Non-configurability. Every ES5 object is born in a *configurable* state. In this state, all of its properties can be updated and deleted. It is possible to *freeze* individual properties of an object, for instance:

```
Object.defineProperty(point,"x",{writable:false,configurable:false});
```

This freezes the `point.x` property so that any attempt to update or delete that property will fail. Once a property is frozen, it remains forever frozen.

An object can evolve to become both non-extensible as well as non-configurable. Such objects are called *frozen* objects. ES5 even introduces an `Object.freeze` primitive that immediately puts an object into the frozen state, by marking the object as non-extensible, and marking all of its properties as frozen. For example, after calling

`Object.freeze(point)`, the `point` object can no longer be extended with new properties, and its existing properties can no longer be deleted or updated.

To detect whether an object is frozen, one can call `Object.isFrozen(point)`. Once the programmer has established that the `point` object is frozen, she can be sure that `point.x` will consistently refer to the same value throughout the object's lifetime.

Abstracting from the details of Javascript, these language invariants have two important characteristics: a) they are *universal*, i.e. they hold for all objects, regardless of their "type" and b) they are *monotonic*: once established on an object, they continue to hold for the entire lifetime of the object.

It are these characteristics that lend the language invariants their power to *locally* reason about objects in a dynamic language such as Javascript. Because the invariants are universal, they hold independent of the type of objects. Because the invariants are monotonic, they hold independent of pointer aliasing [6]. This monotonicity can be observed in the state diagram: there are no operations that take an object from a higher-integrity state to a lower-integrity state[2]. Hence, previously established invariants will continue to hold even if unknown parts of the program retain a reference to the object.

To focus on the essence, throughout the rest of this paper, we will simplify the exposition by combining the two ES5 language invariants (non-extensibility and non-configurability) and only considering non-frozen versus frozen objects. While we will focus on the specific invariants of frozen Javascript objects, the underlying principles should apply to any language-enforced invariant that is universal and monotonic.

2.2 The Problem: Proxies Can't Safely Uphold Language Invariants

Let us now consider how frozen objects interact with proxies. To this end, we will make use of a proposed Proxy API for Javascript on which we reported earlier [7]. Below is the definition of a proxy that simply forwards all intercepted "operations" (such as property access, property assignment, and so on) to a wrapped object:

```
function wrap(target) {
  var handler = {
    get: function(proxy, name) { return target[name]; },
    set: function(proxy, name, value) { target[name] = value; },
    ... // more operations can be intercepted (omitted for brevity)
  };
  var proxy = Proxy.create(handler);
  return proxy;
}
```

The function `Proxy.create` takes as its argument a `handler` object that implements a series of *traps*. Traps are functions that will be invoked by the proxy mechanism when a particular operation is intercepted on the proxy. For instance, the expression `proxy.foo` will be interpreted as `handler.get(proxy, "foo")`. Similarly, the expression `proxy.foo = 42` leads to a call to `handler.set(proxy, "foo", 42)`.

[2] A non-configurable object can be made partially configurable again by adding a new property, but only if the object is extensible. All of its existing frozen properties remain frozen.

From inspecting this code, it is clear that a proxy returned by the `wrap` function will behave the same as its `target` object. The key question is whether `Object.isFrozen(target)` then implies `Object.isFrozen(proxy)`? That is, does the proxy automatically acquire the language invariants of its target object?

The answer unfortunately is no. Whether or not the proxy still upholds the target's invariants depends on the *implementation* of the trap functions, and answering this question is in general undecidable. What is more: the fact that `proxy` is a proxy *for* the `target` object is only implicit in the above program: the `proxy` object does not even possess a direct reference to the `target` object. It just happens to be a proxy *for* it because the trap functions have closed over the `target` variable, and decided to forward the intercepted operations to the object stored in this variable. In fact, it is perfectly reasonable to create proxy objects that don't forward anything to any target object at all (the objects represented by such proxies would be entirely virtual).

Let us illustrate this point by creating an alternative version of `wrap` that probably does not uphold its target object's invariants:

```
function wrapRandom(target) {
  var handler = {
    get: function(proxy, name) { return Math.random(); },
    ...
  };
  var proxy = Proxy.create(handler);
  return proxy;
}
```

If we now call **var** `proxy = wrapRandom(Object.freeze(point))`, passing the frozen `point` object as the `target`, then `proxy.x` may yield a different number each time it is dereferenced. That is hardly the behavior one would expect of a frozen object, so `Object.isFrozen(proxy)` must clearly return false.

It turns out that having `Object.isFrozen` return false for any proxy is the only safe option. The alternative (returning true) would break the invariant that *if* an object is frozen, *then* one can expect property access to consistently return the same value, as the above example demonstrates.

By requiring `Object.isFrozen` to return false for all proxies, we have taken away some of the virtualization power of proxies, by disallowing them to ever virtualize frozen objects. This is necessary to ensure the integrity of the invariant associated with frozen objects ("properties of a frozen object are immutable").

This is an unfortunate outcome, as there certainly exist legitimate use cases where the proxy would want to appear as frozen, and where the proxy handler does behave as a frozen object. One can easily imagine a variant of the `wrap` function that does some form of profiling or contract checking, but otherwise faithfully forwards all operations to its target. Because the proxy cannot faithfully virtualize the frozen invariant of its target, transparency is lost. Such transparency may be crucial in some applications, where objects may need to be substituted by proxies without affecting client behavior.

2.3 The Solution: Invariant Enforcement

To overcome the limitation of proxies to virtualize frozen objects, we modified the initial Proxy API as follows:

- Proxies now refer directly to the target object that they wish to wrap.
- Whenever an operation is intercepted, a trap is called on the handler, as before.
- Before the trap gets to return its result to the client, the proxy verifies whether an invariant is associated with the intercepted operation on the target object.
- If so, the proxy asserts that the result is consistent with the expected (invariant) result. A failed assertion leads to a run-time exception, thus warning the client of an untrustworthy proxy.

Because the proxy now "knows" the target object that the handler wants to virtualize, the proxy has a way of *verifying* whether a) the target has an invariant and b) the trap doesn't violate that invariant. This is the key mechanism by which proxies are enforced to uphold the target object's invariants. We call this mechanism *invariant enforcement*. It is illustrated graphically in Figure 2.

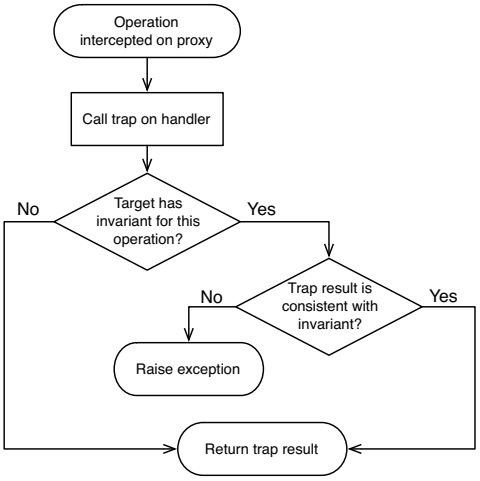


Fig. 2. Invariant enforcement in trustworthy proxies

The `wrap` function from the previous Section can be rewritten using the new API:

```
function wrap(target) {
  var handler = {
    get: function(target, name) { return target[name]; },
    set: function(target, name, value) { target[name] = value; },
    ... // more operations can be intercepted (omitted for brevity)
  };
  var proxy = Proxy(target, handler);
```

```
  return proxy;
}
```

In this version of the API, `Proxy` has become a function of two arguments and takes as its first argument a direct reference to the `target` object for which the proxy will act as a stand-in. The handler traps remain mostly the same, except that instead of the `proxy`, the traps now receive the `target` as their first argument. Figure 3 depicts the relationship between the objects, and shows how operations like property access and update are interpreted by proxies.

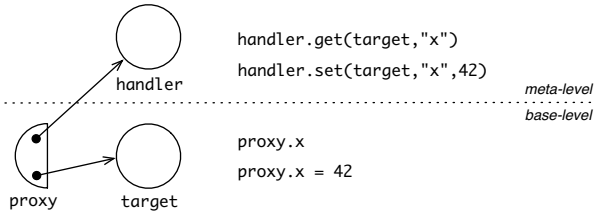


Fig. 3. Relationship between `proxy`, `target` and `handler`

For the proxies generated by this API, it holds that `Object.isFrozen(proxy)` if and only if `Object.isFrozen(target)`. That is, such proxies can faithfully virtualize the frozen invariant of their target. And because of invariant enforcement, the outcome of the operation is actually trustworthy.

How does invariant enforcement work in the specific case of frozen objects and property access? When evaluating the expression `proxy.x`, where `proxy` refers at run-time to a proxy object, one can reason about this property access as if the proxy executed the following code:

```
// inside the proxy, intercepting proxy.x
var value = handler.get(target, "x");
if (Object.isFrozen(target)) {
  var expectedValue = target.x;
  if (expectedValue !== value) {
    throw new Error("frozen invariant violated");
  }
}
return value;
```

Note that invariant enforcement is a two-step process: first the proxy verifies whether there is an invariant to be enforced on the target (in this case: is the target frozen?). If so, then the trap result is verified. It may seem odd that the proxy must first check whether the target has invariants at interception-time. Can it not determine the target's invariants ahead of time? Unfortunately the answer is no, since, in Javascript, objects can come to acquire new invariants at run-time. For instance, an object becomes frozen only *after* a

call to `Object.freeze`. Before that time, the object is not frozen and no invariants need be enforced[3].

To summarize, by redesigning the Proxy API, the apparent tradeoff between proxies and invariants is resolved. Proxies get to virtualize objects with invariants, but only if they can provide a target object that "vouches for" these invariants, such that the result of trap invocations on the handler can be verified. Programmers, secure sandboxes and virtual machines can all continue to rely on the language invariants, regardless of whether they are dealing with built-in objects or proxies.

3 The λ_{TP} Calculus

To study invariant enforcement in more detail, we now turn our attention from Javascript to a minimal calculus named λ_{TP}. While the full Javascript language is fairly large and complex, at its core lies a simple dynamic language with first-class lexical closures and concise object literal notation. This simple core is what Crockford refers to as "the good parts" [8], and it is this core that is modelled by λ_{TP}. At this stage, we should warn the reader that it has *not* been our goal to accurately formalize Javascript. For an accurate formalization of Javascript, we refer to [9].

Our goal is to faithfully model in λ_{TP} the interaction between proxy, target and handler objects as informally discussed in Section 2.3. We first introduce the core calculus with proxies that are not trustworthy. We then add support for trustworthy proxies by revising the semantics.

3.1 Core λ_{TP}

Syntax. λ_{TP} is based on the untyped λ-calculus with strict evaluation rules, extended with constants, records and proxies. It is inspired by the λ_{proxy} calculus of Austin *et al.* [2], which models proxies representing virtual values (see Section 7.2).

Values of the core calculus include constants (strings s and booleans b), functions $\lambda x.e$ and records. Records are either primitive records or proxies. Primitive records are finite mutable maps from strings to values. A primitive record created using the expression $\{\overline{s : e}\}$ declares zero or more initial properties whose value can be retrieved ($e[e]$) or updated ($e[e] := e$). Proxies are created using the expression `proxy` e e where the arguments denote the proxy's target record and handler record respectively.

Records, like Javascript objects, can be frozen. A frozen record cannot be extended with new properties and its existing properties can no longer be updated. Records can be made frozen using the `freeze` r operator, which corresponds to Javascript's `Object.freeze` primitive. The `isFrozen`r operator can be used to test whether the record r is frozen, and models Javascript's `Object.isFrozen(r)` primitive.

In Javascript one can *enumerate* the properties of an object by means of a `for-in` loop. λ_{TP} similarly features a `for` $(x : e)\ e'$ expression that iteratively evaluates the loop body e' with x bound to each property key (a string) of a record e.

[3] However, once an invariant on the target object has been established, a smart implementation could from that point on elide the check for that invariant on future intercepted operations, since the invariant will hold forever after.

The expression `keys` r eagerly retrieves all of the property keys of r. As λ_{TP} does not feature lists or arrays as primitive values, we encode the set of property keys of a record as another record mapping those keys to `true`.

The calculus further includes a `typeof` operator inspired by the corresponding operator in Javascript, revealing the type of a value as a string. The `typeof` operator classifies any value as either a function, a record or a constant.

The λ_{TP} calculus

Syntax

$e ::=$		**Expressions**
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e\ e$	application
	`if` $e\ e\ e$	conditional
	$e = e$	equality test
	$\{\overline{s : e}\}$	record creation
	$e[e]$	record lookup
	$e[e] := e$	record update
	`for` $(x : e)\ e$	enumeration
	`proxy` $e\ e$	proxy creation
	`freeze` e	freezing
	`isFrozen` e	frozen test
	`typeof` e	type test
$c ::= s \mid$ `null` $\mid b$		**Constants**
$b ::=$ `true` $\mid$ `false`		**Booleans**

Syntactic Sugar

$e.x$	$\stackrel{\text{def}}{=}$	$e[\texttt{"x"}]$
$e.x := e'$	$\stackrel{\text{def}}{=}$	$e[\texttt{"x"}] := e'$
$x : e$	$\stackrel{\text{def}}{=}$	$\texttt{"x"} : e$
`let` $x = e\ ;\ e'$	$\stackrel{\text{def}}{=}$	$(\lambda x.e')\ e$
`var` $x = e\ ;\ e'$	$\stackrel{\text{def}}{=}$	`let` $y = \{\}\ ;\ y.x := \theta e\ ;\ \theta e'$ where $\theta = [x := y.x]$
$e\ ;\ e'$	$\stackrel{\text{def}}{=}$	$(\lambda x.e')\ e \quad x \notin FV(e')$
$e\ e'\ e''$	$\stackrel{\text{def}}{=}$	$(e\ e')\ e''$
$\lambda.e$	$\stackrel{\text{def}}{=}$	$\lambda x.e \quad x \notin FV(e)$
$\lambda x, y.e$	$\stackrel{\text{def}}{=}$	$\lambda x.\lambda y.e$
`!` e	$\stackrel{\text{def}}{=}$	`if` e `false true`
`assert` e	$\stackrel{\text{def}}{=}$	`if` e `null` (`null null`)
`keys` e	$\stackrel{\text{def}}{=}$	`let` $r = \{\}\ ;$ `for` $(x : e)\ r[x] :=$ `true` $;\ r$

Semantics. The runtime syntax of λ_{TP} extends expressions with addresses a and enumerations $\mathrm{enum}\ (x : S)\ e$. The latter denote the continuation of an active record enumeration.

λ_{TP} SEMANTICS (UNTRUSTWORTHY PROXIES)

Runtime syntax

$a, t, h \in \mathbf{Address}$
$s \in S \subset \mathbf{String}$

v, w	::=	$c \mid a$	Values
e	::=	$\ldots \mid a \mid \mathrm{enum}\ (x : S)\ e$	Runtime expressions
H	::=	$a \rightarrow_p (\texttt{rec}\ f\ b \mid \texttt{fun}\ x\ e \mid \texttt{pxy}\ t\ h)$	Heaps
f	::=	$s \rightarrow_p v$	Record mapping
E	::=	$\bullet\ e \mid v\ \bullet \mid \texttt{if}\ \bullet\ e\ e \mid \{\overline{s : v}, s : \bullet, \overline{s : e}\}$	Evaluation contexts
	|	$\bullet = e \mid v = \bullet \mid \bullet\,[e] \mid v[\bullet] \mid \texttt{typeof}\ \bullet$	
	|	$\bullet[e] := e \mid v[\bullet] := e \mid v[w] := \bullet \mid \texttt{for}\ (x : \bullet)\ e$	
	|	$\texttt{proxy}\ \bullet\ e \mid \texttt{proxy}\ v\ \bullet \mid \texttt{freeze}\ \bullet \mid \texttt{isFrozen}\ \bullet$	

Reduction rules

$H, \{\overline{s : v}\}$	$\rightarrow H[a \mapsto \texttt{rec}\ f\ \texttt{true}], a$	$a \notin dom(H), f(s) = v$	[ALLOCREC]
$H, \lambda x.e$	$\rightarrow H[a \mapsto \texttt{fun}\ x\ e], a$	$a \notin dom(H)$	[ALLOCFUN]
$H, a\ v$	$\rightarrow H, e[v/x]$	$H(a) = \texttt{fun}\ x\ e$	[APPLY]
$H, \texttt{if true}\ e_1\ e_2$	$\rightarrow H, e_1$		[IFTRUE]
$H, \texttt{if false}\ e_1\ e_2$	$\rightarrow H, e_2$		[IFFALSE]
$H, v = w$	$\rightarrow H, \mathrm{eq}(v, w)$		[EQUAL]
$H, a[s]$	$\rightarrow H, v$	$H(a) = \texttt{rec}\ f\ b, f(s) = v$	[GET]
$H, a[s]$	$\rightarrow H, \texttt{null}$	$H(a) = \texttt{rec}\ f\ b, s \notin dom(f)$	[GETMISSING]
$H, a[s] := v$	$\rightarrow H[a \mapsto \texttt{rec}\ f[s \mapsto v]\ b], v$	$H(a) = \texttt{rec}\ f\ b, b = \texttt{false}$	[SET]
$H, \texttt{for}\ (x : a)\ e$	$\rightarrow H, \mathrm{enum}\ (x : dom(f))\ e$	$H(a) = \texttt{rec}\ f\ b$	[STARTENUM]
$H, \mathrm{enum}\ (x : \emptyset)\ e$	$\rightarrow H, \texttt{null}$		[STOPENUM]
$H, \mathrm{enum}\ (x : S)\ e$	$\rightarrow H, \texttt{let}\ x = s\ ;\ e\ ;$ $\mathrm{enum}\ (x : S')\ e$	$s \in S,\ S' = S \setminus \{s\}$	[STEPENUM]
$H, \texttt{freeze}\ a$	$\rightarrow H[a \mapsto \texttt{rec}\ f\ \texttt{true}], a$	$H(a) = \texttt{rec}\ f\ b$	[FREEZE]
$H, \texttt{isFrozen}\ a$	$\rightarrow H, b$	$H(a) = \texttt{rec}\ f\ b$	[ISFROZEN]
$H, \texttt{typeof}\ a$	$\rightarrow H, \texttt{"function"}$	$H(a) = \texttt{fun}\ x\ e$	[TYPEOFFUN]
$H, \texttt{typeof}\ a$	$\rightarrow H, \texttt{"record"}$	$H(a) = \texttt{rec}\ f\ b$	[TYPEOFREC]
$H, \texttt{typeof}\ c$	$\rightarrow H, \texttt{"constant"}$		[TYPEOFCST]
$H, \texttt{proxy}\ t\ h$	$\rightarrow H[a \mapsto \texttt{pxy}\ t\ h], a$	$a \notin dom(H)$	[ALLOCPXY]
$H, \texttt{typeof}\ a$	$\rightarrow H, \texttt{"record"}$	$H(a) = \texttt{pxy}\ t\ h$	[TYPEOFPXY]
$H, a[s]$	$\rightarrow H, h[\texttt{"get"}]\ t\ s$	$H(a) = \texttt{pxy}\ t\ h$	[GETPXY]
$H, a[s] := v$	$\rightarrow H, h[\texttt{"set"}]\ t\ s\ v\ ;\ v$	$H(a) = \texttt{pxy}\ t\ h$	[SETPXY]
$H, \texttt{freeze}\ a$	$\rightarrow H, h[\texttt{"freeze"}]\ t\ ;\ a$	$H(a) = \texttt{pxy}\ t\ h$	[FREEZEPXY]
$H, \texttt{isFrozen}\ a$	$\rightarrow H, h[\texttt{"isFrozen"}]\ t$	$H(a) = \texttt{pxy}\ t\ h$	[ISFROZENPXY]
$H, \texttt{for}\ (x : a)\ e$	$\rightarrow H, \texttt{for}\ (x : h[\texttt{"enum"}]\ t)\ e$	$H(a) = \texttt{pxy}\ t\ h$	[ENUMPXY]
$H, E[e]$	$\rightarrow H', E[e']$	$H, e \rightarrow H', e'$	[CONTEXT]

A heap H is a partial mapping from addresses a to three types of values: functions, primitive records or proxies. Functions are represented as `fun` $x\ e$ where x is the formal parameter and e is the function body. Records are either primitive records or proxies. Primitive records are represented as `rec` $f\ b$ where f is a partial function from strings to values representing the record's properties and b is a boolean indicating whether or not the record is frozen. Proxies are represented as `pxy` $t\ h$ where t is the address of the proxy's target and h is the address of the proxy's handler (both of which should denote records, not functions).

An evaluation state H, e denotes a heap H and the expression e being evaluated. The rules for the evaluation relation $H, e \rightarrow H', e'$ describe how expressions are evaluated in λ_{TP}.

The [EQUAL] rule describes equality of values v and w in terms of an `eq` primitive, which may be defined as:

$$\mathrm{eq}(v, w) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{true} & \text{if } v \text{ and } w \text{ denote the same constant } c \\ \mathrm{true} & \text{if } v \text{ and } w \text{ denote the same address } a \\ \mathrm{false} & \text{otherwise} \end{cases}$$

This primitive represents identity-equality: constants are equal if and only if they denote the same constant value c; functions, records and proxies are equal if and only if their addresses a are equal.

The rules [GET] and [GETMISSING] together implement record lookup. If the property is not found, `null` is returned (similar to `undefined` being returned for missing properties in Javascript). The [SET] rule implements record update. If a record is updated with a non-existent property, the property is added to the record. Note that the [SET] rule only allows updates on non-frozen records. It is an error to try and add or update a property on a frozen record.

The rules [STARTENUM], [STOPENUM] and [STEPENUM] together implement property enumeration over records. There are two aspects worth noting about our enumeration semantics: first, enumeration is driven by a fixed snapshot of the record's properties. The snapshot includes those properties present when the enumeration starts. Properties added to the record while reducing a `for`-expression will not be enumerated during the same enumeration. Second, the order in which a record's properties are enumerated is left unspecified. This is also true of property enumeration in Javascript.

The rule [FREEZE] shows that the `freeze` operator, applied to a record address a always yields the same address a, but as a side-effect modifies the heap so that the record is now marked frozen, regardless of whether it was already frozen.

3.2 Untrustworthy Proxies in λ_{TP}

A proxy is created using the expression $\mathrm{proxy}\ e\ e'$ where the first expression denotes the proxy's target and the second expression denotes the proxy's handler (both expressions must reduce to an address denoting another record, i.e. either a built-in record or another proxy). λ_{TP} proxies can intercept five operations: record lookup, record update, enumeration, freezing and frozen tests. The signatures of the trap functions are shown below:

$$\begin{aligned}
\texttt{get} &:: target \to string \to result\, value \\
\texttt{set} &:: target \to string \to update\, value \to unit \\
\texttt{freeze} &:: target \to unit \\
\texttt{isFrozen} &:: target \to boolean \\
\texttt{enum} &:: target \to keys\, record
\end{aligned}$$

The rules [GETPXY] and [SETPXY] prescribe that record lookup and record update on a proxy are equivalent to calling that proxy's `get` and `set` traps. Note that the return value of the `set` trap itself is ignored: record update always reduces to the update value v. A similar observation can be made for freezing: the `freeze` operator always returns the frozen object and ignores the result of the `freeze` trap. The `isFrozen` trap, on the other hand, is expected to return a boolean indicating the result of the test.

The rule [ENUMPXY] shows that upon enumerating the properties of a proxy, that proxy's `enum` trap is called. This trap is expected to return a set of property names, encoded as a record. The returned record's properties are subsequently enumerated.

3.3 Language Invariants

While the above reduction rules capture the essence of what it means for a proxy to intercept an operation, they do not aim to uphold any language invariants. In other words: proxies, as currently specified, are not trustworthy. This is easy enough to see: there is nothing stopping a proxy from returning `true` from its `isFrozen` trap, while still reporting different values over time for its properties via its `get` trap.

In this Section, we explicitly spell out the invariants of the λ_{TP} calculus with respect to frozen *primitive* records. In the following Section, we then revise the proxy reduction rules such that proxies obey the same invariants as primitive records, thus making proxies trustworthy. The invariants of frozen primitive records are as follows:

I1 The properties of a frozen record are immutable. If $\texttt{isFrozen}\, r = \texttt{true}$ then $r[s]$ always reduces to the same result value v. This also implies that if r does not have a property named s, $r[s]$ will always reduce to `null`.

I2 If the $\texttt{freeze}\, r$ operator returns successfully, r is guaranteed to be frozen.

I3 Freezing is monotonic: once $\texttt{isFrozen}\ r = \texttt{false}$, it remains `false` thereafter. In other words: once frozen, a record remains forever frozen.

I4 Enumerating properties using a `for`-loop over a frozen record r always enumerates the same set of properties. That is, it enumerates at least all properties defined on r, and it does not enumerate any properties that do not exist on r.

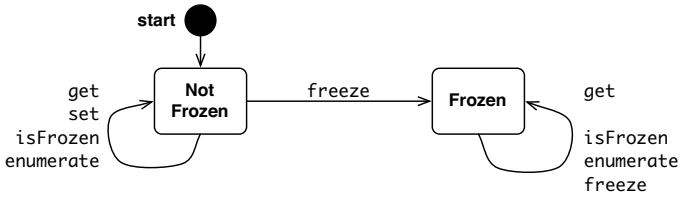


Fig. 4. State chart depicting the valid states of a λ_{TP} record

Figure 4 depicts a state chart illustrating the state of a record, and the effect of interceptable operations on records on that state.

3.4 Trustworthy Proxies in λ_{TP}

In the semantics described thus far, proxies are *untrustworthy*, meaning that they may violate the above invariants of primitive records. Below, we introduce a set of updated reduction rules, which turn λ_{TP} proxies into *trustworthy* proxies.

λ_{TP} **SEMANTICS (TRUSTWORTHY PROXIES)**

$$
\begin{array}{rll}
H, a[s] \rightarrow H, \texttt{let}\ x = h[\texttt{"get"}]\ t\ s\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{getPxy'}] \\
\quad \texttt{if}\ (\texttt{isFrozen}\ t) & & \\
\quad\quad (\texttt{assert}\ x = t[s]\ ;\ x) & & \\
\quad\quad x & & \\
H, a[s] := v \rightarrow H, h[\texttt{"set"}]\ t\ s\ v\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{setPxy'}] \\
\quad \texttt{assert}\ (!\ \texttt{isFrozen}\ t)\ ;\ v & & \\
H, \texttt{for}\ (x : a)\ e \rightarrow H, \texttt{let}\ y = h[\texttt{"enum"}]\ t\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{enumPxy'}] \\
\quad \texttt{if}\ (\texttt{isFrozen}\ t) & & \\
\quad\quad \texttt{sameKeys}\ y\ (\texttt{keys}\ t) & & \\
\quad\quad \texttt{null}\ ; & & \\
\quad \texttt{for}\ (x : y)\ e & & \\
H, \texttt{freeze}\ a \rightarrow H, h[\texttt{"freeze"}]\ t\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{freezePxy'}] \\
\quad \texttt{assert}\ (\texttt{isFrozen}\ t)\ ;\ a & & \\
H, \texttt{isFrozen}\ a \rightarrow H, \texttt{let}\ x = h[\texttt{"isFrozen"}]\ t\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{isFrozenPxy'}] \\
\quad \texttt{assert}\ (x = \texttt{isFrozen}\ t)\ ;\ x & &
\end{array}
$$

The rule [GETPXY'] includes a post-condition that asserts whether the return value of the `get` trap corresponds to the target's value for the same property s, but only if the target is frozen. This assertions contributes to upholding invariant **I1**. Note that the proxy target t may itself be a proxy, in which case the expression $t[s]$ in the assertion will recursively trigger the [GETPXY'] rule, eventually bottoming out when the target is a primitive record.

The rule [SETPXY'] includes a post-condition that asserts that the target is not frozen. Again, this assertion ensures invariant **I1**.

The rule [FREEZEPXY'] includes a similar post-condition, this time testing whether the target is indeed frozen, if the `freeze` trap returned successfully. Clients that call `isFrozen r` expect `r` to be frozen afterwards. This guarantees invariant **I2**.

The rule [ISFROZENPXY'] inserts a post-condition that verifies whether the return value of the `isFrozen` trap corresponds to the current state of the target. Any discrepancy in the result could confuse client code: if the `isFrozen` trap is allowed to return `true` while wrapping a non-frozen target, clients would perceive the proxy as frozen while its `get` trap could still return arbitrary values, thus breaking invariant **I1**. The other way around, if the trap is allowed to return `false` while wrapping a frozen target, it may break invariant **I3** if it previously already returned `true` from its `isFrozen` trap.

The rule [ENUMPXY'] inserts a post-condition, ensuring that if the proxy wraps a frozen target, the returned set of to-be-enumerated properties corresponds to the set of properties of the target itself. This guarantees invariant **I4**. `sameKeys` is defined as follows:

$$\begin{aligned}\texttt{sameKeys}\, r_1, r_2 \stackrel{\text{def}}{=} \texttt{for}\,(x_1 : r_1)\,\texttt{assert}\, r_2[x_1] = \texttt{true}\,;\\ \texttt{for}\,(x_2 : r_2)\,\texttt{assert}\, r_1[x_2] = \texttt{true}\end{aligned}$$

`sameKeys` checks whether two records representing sets of properties denote the same set of property keys. One can think of the first `for`-loop as checking whether all properties that the proxy enumerated are indeed properties of the frozen target, and of the second `for`-loop as checking whether the proxy did indeed enumerate all properties of the frozen target.

Note that the validity of all pre and post-condition checks depends also on the fact that the assertions compare the trap result against the expected value using the = operator, and that proxies cannot intercept this operator. Thus, proxies cannot directly influence the outcome of the assertions.

4 Access Control Wrappers

We now put trustworthy proxies to work by using them to build access control wrappers. Such wrappers typically perform a set of dynamic checks upon intercepting certain operations, but otherwise try to be as transparent as possible to client objects. If the check succeeds, the wrapper often simply forwards the intercepted operation to the wrapped object. Using trustworthy proxies, we can build access control wrappers that uphold the invariants of wrapped target objects, further increasing transparency for clients.

4.1 Revocable References

A revocable reference is a simple type of access control wrapper. Say an object `alice` wants to hand out to `bob` a reference to `carol`. `carol` could represent a precious resource, and for that reason `alice` may want to limit the lifetime of the reference she hands out to `bob`, which she may not fully trust. In other words, `alice` wants to have the ability to revoke `bob`'s access to `carol`. Once revoked, `bob`'s reference to `carol` should become useless.

One can implement this pattern of access control by wrapping `carol` in a forwarding proxy that can be made to stop forwarding. This is also known as the *caretaker* pattern [10]. In the absence of language-level support for proxies, the programmer is forced to write a distinct caretaker for each type of object to be wrapped. Proxies enable the programmer to abstract from the specifics of the wrapped object's interface and instead write a *generic* caretaker. Using such a generic caretaker abstraction, `alice` can hand out a revocable reference to `bob` as follows:

```
var carol = {...};
// caretaker is a tuple consisting of a proxy reference, and a revoke function
var caretaker = makeCaretaker(carol);
```

```
var carolproxy = caretaker.ref; // a proxy for carol, which alice can give to bob
bob.use(carolproxy);
// later, alice can revoke bob's access...
caretaker.revoke(); // carolproxy is now useless
```

A key point is that as long as the caretaker is not revoked, the proxy is sufficiently transparent so that `bob` can use `carolproxy` as if it were the real `carol`. There is no need for `bob` to change the way he interacts with `carol`. Indeed, if `bob` has no other, direct, reference to `carol`, `bob` is not even able to tell that `carolproxy` is only a proxy for `carol`.

Below is an implementation of the `makeCaretaker` abstraction in λ_{TP}:

$$
\begin{array}{l}
\mathrm{makeCaretaker} \stackrel{\mathrm{def}}{=} \lambda x. \\
\ \mathtt{var}\ \mathrm{revoked} = \mathtt{false}; \\
\quad \{\mathrm{ref} : \mathtt{proxy}\ x\ \{ \\
\qquad \mathrm{get} : \lambda t, s.\mathtt{assert}\ (!\mathrm{revoked})\ ; t[s] \\
\qquad \mathrm{set} : \lambda t, s, v.\mathtt{assert}\ (!\mathrm{revoked})\ ; t[s] := v \\
\qquad \mathrm{enum} : \lambda t.\mathtt{assert}\ (!\mathrm{revoked})\ ; \mathtt{keys}\ t\ \} \\
\qquad \mathrm{freeze} : \lambda t.\mathtt{assert}\ (!\mathrm{revoked})\ ; \mathtt{freeze}\ t \\
\qquad \mathrm{isFrozen} : \lambda t.\mathtt{assert}\ (!\mathrm{revoked})\ ; \mathtt{isFrozen}\ t\ \} \\
\quad \mathrm{revoke} : \lambda.\mathrm{revoked} := \mathtt{true}\}
\end{array}
$$

The argument x to `makeCaretaker` is assumed to be a record. The function returns a record `r` that pairs a proxy `r.ref` with an associated function `r.revoke`. Both share a privately scoped `revoked` boolean that signifies whether or not the proxy was previously revoked. The proxy's handler implements all traps by first verifying whether the reference is still unrevoked. If this is the case, it forwards each operation directly to the wrapped target record t.[4]

A limitation of the above caretaker abstraction is that values exchanged via the caretaker are themselves not recursively wrapped in a revocable reference. For example, if `carol` defines a method that returns an object, she exposes a direct reference to that object to `bob`, circumventing `alice`'s caretaker. The returned object may even be a reference to `carol` herself (e.g. by returning `this` from a method in Javascript). The abstraction discussed in the following section addresses this issue.

4.2 Membranes: Transitively Revokable References

A membrane is an extension of a caretaker that transitively imposes revocability on all references exchanged via the membrane [11].

One use case of membranes is the safe composition of code from untrusted third parties on a single web page (so-called "mash-ups"). Assuming the code is written in a safe subset of Javascript, such as Caja [12], loading the untrusted code inside such a membrane can fully isolate scripts from one another and from their container page. Revoking the membrane around such a script then renders it instantly powerless.

[4] We take the notational liberty of using names like t,s and v for trap parameters, which hint at the parameters' type, rather than using strict variable names like x, y and z.

The objective of the membrane is to fully keep the object graph g created by the untrusted code isolated from the object graph g' of the containing page. When creating a membrane, one usually starts with a single object that forms the "entry point" into g. At the point when the membrane is created, it is usually assumed that apart from a single reference to the entry point of g, g and g' are otherwise fully isolated (i.e. there are no other direct references from any objects in g to any objects in g' and vice versa). If this is not the case, then the membrane will not be able to fully enclose and isolate g.

The following example demonstrates the transitive effect of a membrane. The prefix `wet` identifies objects initially inside of the membrane, while `dry` identifies revokable references outside of the membrane designating wet objects.

```
var wetA = { x: 1 }
var wetB = { y: wetA }
var membrane = makeMembrane(wetB) // wetB acts as the entry point
var dryB = membrane.ref // a proxy for wetB
var dryA = dryB.y // references are transitively wrapped
dryA.x            // returns 1, constants are not wrapped
membrane.revoke() // revokes all dry references at once
dryB.y            // error: revoked
dryA.x            // error: revoked
```

The interface of a membrane is the same as that of a caretaker. Its implementation in λ_{TP} is shown in Figure 5. A membrane consists of one or more wrappers. Every such wrapper is created by a call to the `wrap` function. All wrappers belonging to the same membrane share a single `revoked` variable. Assigning the variable to `false` instantaneously revokes all of the membrane's wrappers.

```
makeMembrane ≝ λx.
 var revoked = false;
   let wrap = λy.
    if typeof y = "constant"
       y
       if typeof y = "function"
          λz.assert (!revoked) ; wrap (y (wrap z))
          proxy y {
            get : λt, s.assert (!revoked) ; wrap t[s]
            set : λt, s, v.assert (!revoked) ; t[s] := (wrap v)
            freeze : λt.assert (!revoked) ; freeze t
            isFrozen : λt.assert (!revoked) ; isFrozen t
            enum : λt.assert (!revoked) ; keys t } ;
    {ref : wrap x
     revoke : λ.revoked := true}
```

Fig. 5. Membranes in λ_{TP}

The `wrap` function does a case-analysis on y based on the three types of values in λ_{TP}: constants are passed through a membrane unwrapped; functions are wrapped as functions that transitively wrap their argument and return value; and records are wrapped using proxies.

Although this implementation does not distinguish them, there are two "directions" in which a value can flow across the membrane: the argument z to a wrapped function and the argument v to the `set` trap are inbound, while the return value of a wrapped function and the return value of the `get` trap are outbound. In this version, both inbound and outbound values get wrapped without distinction.

Note that because the membrane faithfully forwards the `freeze` and `isFrozen` operations (as long as it is not revoked), clients on either side of the membrane can inspect whether or not the wrapped object is frozen and act accordingly. Because proxies are trustworthy, clients can have complete confidence in the outcome of the `freeze` and `isFrozen` operators, even when a membrane is interposed.

4.3 Membranes and Frozen Objects

The above membrane implementation works fine except for one important detail, which surfaces when a frozen record crosses the membrane. Things go wrong when code on either the dry (outside) or the wet (inside) side of the membrane tries to lookup a property $r[s]$ on a wrapper r for a frozen record t. Instead of getting back a transitively wrapped value, the program will trip on an assertion.

Recall that the invariant enforcement mechanism inserts a post-condition check upon evaluating $r[s]$ (rule [GETPXY']), testing whether its return value is equal to $t[s]$. However, the `get` trap of the above membrane abstraction always returns a fresh wrapper for the value $t[s]$ of the wrapped, frozen target t. Unless $t[s]$ is a constant, the check fails since the trap is returning a proxy for $t[s]$, not $t[s]$ itself.

To circumnavigate this issue, rather than letting the proxy wrapper directly wrap the target on the other side of the membrane, we let it wrap a *shadow target*, a dummy – initially empty – record that is initially not frozen. Figure 6 shows the initial state of such a membrane proxy.

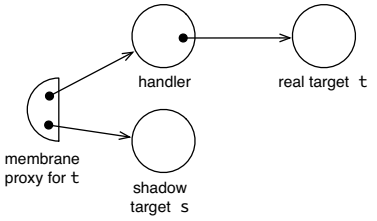


Fig. 6. A membrane proxy with a shadow target

The purpose of the shadow target is to store wrapped properties of a frozen target. This way, when the handler returns a wrapper for the original target property from the `get` trap, the proxy checks the result against the shadow target, not the original target. Since the shadow target only contains wrapped properties, the invariant check will succeed. Thus, the membrane now operates correctly on frozen records.

We will henceforth refer to the shadow target simply as "the shadow", and to the real target as "the target". Having introduced a shadow, we now have two records (the

shadow and the target) that can be either frozen or non-frozen. We must ensure that this "frozen state" of both records remains synchronized. Otherwise, consider a membrane wrapper with a non-frozen shadow but a frozen target: when a client asks whether such a wrapper is frozen, the wrapper cannot answer `true`, as the proxy will check the answer against the state of the shadow, which is non-frozen.

We employ the following strategy to keep the shadow and the target in sync: as long as the real target is not frozen, the shadow is also not frozen. In this case, no invariants on the proxy are enforced, and the proxy is free to return non-identical, wrapped values from its `get` trap. There is no need to store these wrapped values on the shadow.

The first time that a proxy must reveal to a client that it is frozen[5], the proxy first *synchronizes* the state of the shadow with that of the target. It does this by defining a wrapped property on the shadow for each property on the target, and then freezing the shadow. Once the shadow is frozen, every time a property is accessed, the value returned is the wrapper defined on the shadow, not the original value.

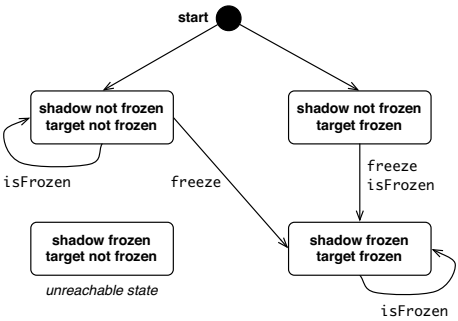


Fig. 7. State chart depicting the states of a membrane wrapper with a shadow target

Figure 7 shows a state chart with the allowable states of a membrane proxy with a shadow target. The four possible states are determined by whether or not either shadow target or real target are frozen. A transition to the lower right state always implies a synchronization of the shadow target with the real target before freezing the shadow.

Figure 8 depicts the state of the proxy after synchronization with a target with a single `foo` property.

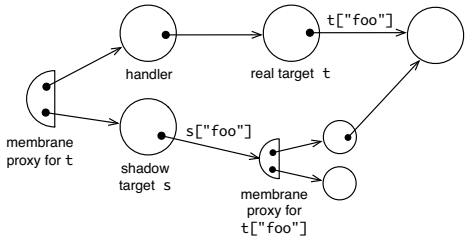


Fig. 8. After synchronization, the shadow caches wrapped values of the target

[5] A proxy must reveal that it is frozen when intercepting `freeze` or when the proxy must answer `true` in response to an `isFrozen` test because the real target is frozen.

```
makeMembrane ≝ λx.
 var revoked = false;
   let wrap = λy.
    if typeof y = "constant"
       y
       if typeof y = "function"
           λz.assert (!revoked) ; wrap (y (wrap z))
           proxy {} {
             get : λt_s, s.assert (!revoked) ; if (isFrozen t_s) t_s[s] (wrap y[s])
             set : λt_s, s, v.assert (!revoked) ; y[s] := (wrap v)
             freeze : λt_s.assert (!revoked) ; freeze y ; (sync t_s y)
             isFrozen : λt_s.assert (!revoked) ; if (isFrozen y) (sync t_s y ; true) false
             enum : λt_s.assert (!revoked) ; keys y } ;
   {ref : wrap x
    revoke : λ.revoked := true}

sync ≝ λt_s, t_r.if (!isFrozen t_s)
                    (for (s : t_r) t_s[s] := wrap t_r[s] ; freeze t_s)
                    null
```

Fig. 9. Membranes with shadow target in λ_{TP}

A membrane making use of this synchronization strategy between shadow and target is shown in Figure 9. Note that the first argument passed to **`proxy`** is an empty object $\{\}$ (the shadow) and not the real target y. Consequently, the first argument passed to each trap t_s denotes the shadow target. In the definition of "sync", t_s similarly stands for the shadow target while t_r stands for the real target.

To conclude this Section, we have shown how membranes can be interposed between two object graphs while preserving the frozen state of objects across both sides of the membrane *and* that operate correctly on frozen objects. The key idea is for a membrane proxy to not wrap the real target object directly, but rather to wrap a shadow target that can store wrapped properties of the target object. If the proxy is itself asked to become frozen, or to reveal that it is frozen via the `isFrozen` operator, it first synchronizes the state of its shadow before proceeding. This ensures that no invariant enforcement assertions will fail on the proxy.

4.4 Identity-Preserving Membranes

In the previous Sections, we presented minimal yet useful implementations of the membrane pattern. Nonetheless, these implementations still have a number of issues regarding object identity that practical membrane implementations can and should address.

First, for non-frozen records, the wrappers are not cached, so if a record is passed through the same membrane twice, clients will receive two distinct wrappers, even though both wrap the same record. Consider the following λ_{TP} expression:

$$\texttt{let}\ x = \text{makeMembrane}(\{s : \{\}\}).\text{ref}\ ;\ (x[s] = x[s])$$

This expression will always reduce to `false` since each occurrence of $x[s]$ reduces to a fresh wrapper for the value of the s property.

Second, no distinction is made between the opposite directions in which a record can cross a membrane. If a record is passed through the membrane in one direction, and then passed through the membrane again in the opposite direction, one would expect to retrieve the original object. However, consider the following λ_{TP} expression, where x denotes a membraned proxy and v denotes a record:

$$x[s] := v\ ;\ \texttt{let}\ y = x[s]\ ;\ (v = y)$$

This expression will also always reduce to `false` since y will be a wrapper for a wrapper for v. In $x[s] := v$, v is wrapped when it crosses the membrane inwards. Then, in $y = x[s]$, the wrapped value is wrapped again when it crosses the membrane outwards. It would be better for these crossings to cancel each other out instead.

These limitations can be addressed by having the membrane maintain extra mappings that map proxies to their wrapped values and vice versa. The details are beyond the scope of this paper. Suffice it to say that in practice the above two problems are addressable in ECMAScript using WeakMaps[6], which are identity hashmaps with the weak referencing semantics of Ephemerons [13].

5 Discussion

The cost of transparent invariants. Membranes, while being a very generic abstraction, are a useful case study because they aim to *transparently* interpose between two object graphs. For the membrane to be adequately transparent, it must be able to accurately uphold invariants across the membrane.

The invariant enforcement mechanism complicates membranes, as it prevents a membrane proxy from directly exposing the properties of a frozen target object as wrapped properties. Instead, we introduced a shadow target to hold the wrapped properties. This has two costs: first, there is the memory overhead of duplicating all the properties on the shadow. Second, operations that modify or expose the state of an object (i.e. `freeze` and `isFrozen`) must explicitly synchronize the state of the shadow and the real target.

While these overheads are not to be underestimated, we believe they are manageable in practice. First, we expect the dominant operations on objects to be `get` and `set`, not `freeze` and `isFrozen`. Second, wrapped properties are only defined on the shadow lazily, i.e. only when the proxy is about to reveal that it is frozen for the first time. If a proxy is never tested for frozenness, the shadow is never even used. Only when a proxy is revealed as frozen must transitive wrappers for the target properties be defined on the shadow.

Garbage collection. In Section 4.1 we introduced revocable references. One of the primary use cases of such references is to facilitate memory management by reducing the

[6] See `http://wiki.ecmascript.org/doku.php?id=harmony:weak_maps`.

risk of memory leaks. The idea is that if objects only hold a revocable reference to a certain resource object, then revoking that reference instantly removes all live references to the resource, allowing it to be garbage collected.

In our earlier Proxy API [7], a proxy did not store an implicit reference to a target object. Rather, it was the handler's responsibility to explicitly manage the reference to the target object, as shown in the first code snippet in Section 2.2. Upon revoking the proxy, that reference would be nulled out, allowing the garbage collector to collect the target object.

In the trustworthy Proxy API introduced here, the proxy holds an implicit reference to the target object which is not under programmer control. The proxy needs this reference to perform its invariant checks. Unfortunately this also implies that there is no way for the handler to null out this reference when the proxy is revoked. Hence, the revocable references introduced in Section 4.1 cannot be used for memory management purposes.

In Javascript, we solved this problem by providing revocable proxies as a primitive abstraction. This allows the proxy implementation to null out the references to its target and handler upon revocation.

Handlers and targets as proxies. In Javascript, as in λ_{TP}, both the target and the handler of a proxy can themselves be proxies. The fact that a handler can itself be a proxy is a property that we have found useful in writing highly generic proxy abstractions. For a concrete example, we refer to our prior work [7].

The fact that the target of a proxy can itself be a proxy raises questions about the validity of our invariant enforcement mechanism. If the target is itself a proxy, might it not be able to return inconsistent results so as to mislead the invariant checks? After all, invariant enforcement hinges on the fact that the target object cannot lie about what invariants it upholds. Fortunately, this is not the case. We sketch an informal proof by induction on the target of a proxy.

First, we note that any chain of proxy-target links must be finite and non-cyclic: the target of a proxy must already exist before the proxy is created. It is not possible to initialize the target of a newborn proxy to that proxy itself.

In the base case, a proxy's target is a regular non-proxy object. Non-proxy objects by definition uphold language invariants, so the proxy can faithfully query the target for its invariants. Hence, the handler will not be able to violate reported invariants.

For the inductive step, consider a proxy a whose target is itself a proxy b. By the induction hypothesis, b cannot violate the invariants of its own target, so that a can faithfully query b for its invariants. Hence, a's handler will not be able to violate b's reported invariants.

Alternatives to runtime assertions. To make proxies trustworthy, we currently rely on run-time post-condition assertions on the return value of trap functions. Some of these assertions, most notably those for the `keys` trap, are relatively expensive, which has prompted us to look into alternative designs for achieving trustworthy proxies.

One design alternative (proposed to us by E. Dean Tribble) is to *ignore* the return value of trap functions altogether, and instead always forward the intercepted operation to the target object *after* having invoked the trap. The outcome of the operation on the

proxy is then guaranteed to be the same as the outcome of the operation on the target, so invariants are preserved. This essentially turns traps into callbacks (event handlers) that get notified right before performing the operation on the target. Since the notification takes place *before* forwarding the operation, the trap may still indirectly determine the outcome of the intercepted operation by manipulating the target object.

While this design avoids run-time invariant checks, it has the downside of making it even harder to express virtual object abstractions, as the virtual object is forced to define a concrete property on the target object for every virtual property that is accessed. As we have not yet fully explored this design alternative, we will refrain from going into more detail here. It does teach us that our proposed design is not the only way of achieving trustworthy proxies, and that there is a broader design space with trade-offs to explore.

6 Availability

Trustworthy proxies are known as "direct proxies" in Javascript. There currently exist two implementations of direct proxies. The first is a native implementation in Firefox 18. The second is a self-hosted implementation via the `reflect.js` library[7]. `reflect.js` is a small Javascript library implemented by the first author. The library implements trustworthy proxies on top of our previously proposed Proxy API for Javascript [7] which is available natively in Firefox and Chrome. The library essentially uses untrustworthy proxies to implement trustworthy proxies in Javascript itself. An implementation of membranes that preserve invariants is shipped with the library.

7 Related Work

For an overview of related Proxy and reflection APIs, we refer to our earlier work [7]. Here, we specifically discuss related work on invariant enforcement in Proxy APIs.

7.1 Chaperones and Impersonators

Chaperones and impersonators are a recent addition to Racket [14]. They are the run-time infrastructure for Racket's contract system on higher-order, stateful values.

Chaperones and impersonators are both kinds of proxies. One difference is that impersonators can only wrap *mutable* data types, while chaperones can wrap both mutable and immutable data types. A second difference is that chaperones can only further *constrain* the behavior of the value that it wraps. When a chaperone intercepts an operation, it must either raise an exception, return the same result that the wrapped target would return, or return a chaperone for the original result. Impersonators, on the other hand, are free to change the value returned from intercepted operations.

Chaperones are similar to trustworthy proxies, in that they restrict the behavior of a wrapper. A trustworthy proxy that wraps a frozen object is constrained like a chaperone. It is actually more constrained, since a chaperone is allowed to return a chaperone for the original value, while trustworthy proxies are not allowed to return a proxy for the

[7] See `http://github.com/tvcutsem/harmony-reflect`.

value of a frozen property. Conversely, as long as the wrapped object is non-frozen, trustworthy proxies are like impersonators and may modify the result of operations.

There are important differences between chaperones and trustworthy proxies, however. First, as chaperones are allowed to return *wrappers* for the original values, *even* for "immutable" data structures, they avoid the overhead of the shadow target technique that we employed for membranes. However, this comes at the cost of weakening the meaning of "immutable": accessing the elements of an immutable vector, wrapped with a chaperone, may yield new wrappers each time an element is accessed. Behaviorally, the element will be the same (modulo exceptions), but structurally it may have a different identity. Thus, the invariant that immutable vectors must always return an *identical* value upon access is weakened.

Second, trustworthy proxies may *both* be a chaperone and an impersonator at the same time. In Racket, a value can be classified as (permanently) mutable or immutable. This distinction cannot be made in Javascript: not only can objects be "half-mutable" (cf. the "partially configurable" state in Figure 1), their mutability constraints can also change at runtime (e.g. by calling `Object.freeze`). Hence, upon wrapping a Javascript object, one cannot decide at that time whether to wrap it with a chaperone or an impersonator. That is why trustworthy proxies must use dynamic checks to test whether to behave as an impersonator (no invariant checks required) or as a chaperone-like proxy (with restricted behavior).

7.2 Virtual Values

Starting from our initial Proxy API [7], Austin et al. have recently introduced a complementary Proxy API for virtualizing primitive *values* [2]. They focus on creating proxies for objects normally thought of as primitive values, such as numbers and strings. They highlight various use cases, such as new numeric types, delayed evaluation, taint tracking, contracts, revokable membranes and units of measure.

Like trustworthy proxies, virtual values are proxies with a separate handler. The handler for a virtual value proxy provides a different set of traps. A virtual value can intercept unary and binary operators applied to it, being used as a condition in an `if`-test, record access and update, and being used as an index into another record.

An important difference between the trustworthy proxies API as presented here, and the virtual values API, is that the latter provides a general `isProxy` primitive that tests whether or not a value is a proxy. Our API does not introduce such an explicit test because it breaks transparent virtualization. As a design decision, we do not want clients to know or care that they are dealing with proxies for other objects.

For virtual values, the `isProxy` primitive was added to enable clients to defend themselves against malicious virtual values, such as mutable strings in a language that otherwise only has immutable strings. The idea is for clients to defend themselves against malicious proxies by explicitly testing whether or not the the object they are interacting with is a proxy. Virtual value proxies have no invariant enforcement.

Trustworthy proxies provide an alternative solution. As trustworthy proxies cannot violate language invariants, the better way for clients to protect themselves against erratic object behavior is to test whether an object is frozen, rather than testing whether it is a proxy.

7.3 Java Proxies

The `java.lang.reflect.Proxy` API [15], introduced in Java 1.3, enables one to intercept method invocations on instances of interface types. Like the Proxy API sketched here, a Java proxy has an associated handler object (known as an `InvocationHandler`) to trap invocations. Unlike trustworthy proxies, Java proxies do not necessarily wrap a target object.

The Java Proxy API only supports proxies for interface types, not class types. As a result, proxies cannot be used in situations where code is typed using class types rather than interface types, limiting their general applicability. Eugster [1] describes an extension of Java proxies that works uniformly with instances of non-interface classes, which, in addition to method invocation, can also trap field access and assignment.

Java proxies have little need for elaborate invariant enforcement. This is partly because Java provides no operations that change the structure of objects (the fields and methods of an object are fixed), and partly because Java proxies do not virtualize field access, so the notion of virtualizing a `final` field does not arise.

However, there is one invariant on Java proxies that is maintained via a runtime check: the runtime type of the return value of the `InvocationHandler`'s `invoke` method must be compatible with the statically declared return type of the intercepted method. If the handler violates this invariant, the proxy implementation throws a `ClassCastException`. This makes Java proxies trustworthy when it comes to the return type of intercepted method invocations.

8 Conclusion

Proxies are a useful part of reflection APIs that enable a variety of use cases, from generic wrapper abstractions such as membranes and higher-order contracts, to virtual object abstractions such as remote object references and lazy initialization. In a language with both proxies and language invariants, these features interact. If a proxy is allowed to emulate an object with language invariants, the question arises whether these invariants are still enforced by the language.

We presented *trustworthy* proxies, which are proxies that can wrap objects with (universal and monotonic) language invariants, where these invariants are enforced through runtime checks by the proxy mechanism. This ensures that proxies cannot circumvent these invariants, such that developers and the VM itself can continue to rely on these invariants even in the presence of proxies.

We explored the need for trustworthy proxies in the context of Javascript, and presented a formal semantics for λ_{TP}, an extension of the λ-calculus including trustworthy proxies. We have shown how abstractions such as transitively revocable references (i.e. membranes) can be built using trustworthy proxies, achieving a transparent interposition between two object graphs which accurately represents language invariants on both sides of the membrane.

Acknowledgements. We thank the members of the ECMA TC-39 committee and the `es-discuss` community for their detailed feedback on this work.

References

1. Eugster, P.: Uniform proxies for java. In: OOPSLA 2006: Proceedings of the 21st Annual Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 139–152. ACM, NY (2006)
2. Austin, T.H., Disney, T., Flanagan, C.: Virtual values for language extension. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 921–938. ACM, New York (2011)
3. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2002, pp. 48–59. ACM, New York (2002)
4. Pratikakis, P., Spacco, J., Hicks, M.: Transparent proxies for java futures. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, pp. 206–223. ACM, New York (2004)
5. ECMA International: ECMA-262: ECMAScript Language Specification. 5th edn. ECMA, Geneva, Switzerland (December 2009)
6. Fähndrich, M., Leino, K.R.M.: Heap monotonic typestates. In: International Workshop on Aliasing, Confinement and Ownership (IWACO 2003), pp. 58–72 (2003)
7. Van Cutsem, T., Miller, M.S.: Proxies: design principles for robust object-oriented intercession APIs. In: Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, pp. 59–72. ACM (2010)
8. Crockford, D.: Javascript: The Good Parts. O'Reilly (2008)
9. 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)
10. Redell, D.D.: Naming and Protection in Extensible Operating Systems. PhD thesis, Department of Computer Science, University of California at Berkeley (November 1974)
11. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, John Hopkins University, Baltimore, Maryland, USA (May 2006)
12. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized javascript (June 2008), `http://tinyurl.com/caja-spec`
13. Hayes, B.: Ephemerons: a new finalization mechanism. In: Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 1997, pp. 176–183. ACM, New York (1997)
14. Strickland, T.S., Tobin-Hochstadt, S., Findler, R.B., Flatt, M.: Chaperones and impersonators: run-time support for reasonable interposition. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 943–962. ACM, New York (2012)
15. Blosser, J.: Explore the Dynamic Proxy API (2000), `http://www.javaworld.com/jw-11-2000/jw-1110-proxy.html`

Java$_{UI}$: Effects for Controlling UI Object Access

Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman

University of Washington
{csgordon,wmdietl,mernst,djg}@cs.washington.edu

Abstract. Most graphical user interface (GUI) libraries forbid accessing UI elements from threads other than the UI event loop thread. Violating this requirement leads to a program crash or an inconsistent UI. Unfortunately, such errors are all too common in GUI programs.

We present a polymorphic type and effect system that prevents non-UI threads from accessing UI objects or invoking UI-thread-only methods. The type system still permits non-UI threads to hold and pass references to UI objects. We implemented this type system for Java and annotated 8 Java programs (over 140KLOC) for the type system, including several of the most popular Eclipse plugins. We confirmed bugs found by unsound prior work, found an additional bug and code smells, and demonstrated that the annotation burden is low.

We also describe code patterns our effect system handles less gracefully or not at all, which we believe offers lessons for those applying other effect systems to existing code.

1 Introduction

Graphical user interfaces (GUIs) were one of the original motivations for object-oriented programming [1], and their success has made them prevalent in modern applications. However, they are an underappreciated source of bugs. A Google search for "SWT invalid thread access" — the exception produced when a developer violates multithreading assumptions of the SWT GUI framework — produces over 150,000 results, including bug reports and forum posts from confused developers and users. These bugs are user-visible, and programs cannot recover from them. Typically, they terminate program execution. Furthermore, these bugs require non-local reasoning to locate and fix, and can require enough effort that some such bugs persist for years before being fixed [2]. Because these bugs are common, severe, and difficult to diagnose, it is worthwhile to create specialized program analyses to find errors in GUI framework clients.

A typical user interface library (such as Java's Swing, SWT, and AWT toolkits, as well as toolkits for other languages) uses one UI thread running a polling event loop to handle input events. The library assumes that all updates to UI elements run on the UI thread. Any long-running operation on this thread would prevent the UI from responding to user input, and for this reason the UI library includes methods for running tasks on *background threads*. A background thread runs independently from the UI thread, and therefore it does not block UI interactions, but it is also restricted in its interactions with the UI. The UI library also provides mechanisms for background threads to update UI elements, primarily by executing a closure on the UI thread, synchronously or asynchronously. For SWT, these correspond to the static methods `syncExec` and `asyncExec`

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 179–204, 2013.

respectively, in the `Display` class. Each accepts a `Runnable` instance whose `run()` method will be executed (synchronously or asynchronously) on the UI thread. For example, a method to update the text of a label from a background thread might look like this:

```
private final JLabel mylabel;
...
public void updateText(final String str) {
    Display.syncExec(new Runnable {
        public void run() { mylabel.setText(str); }
    });
}
```

The separation between the UI and background threads gives several advantages to the UI library implementor:

- Forced atomicity specifications: background threads must interact with the UI only indirectly through the closure-passing mechanism, which implicitly specifies *UI transactions*. Because all UI updates occur on one thread, each transaction executes atomically with respect to other UI updates.
- Minimal synchronization: Assuming clients of the UI library never access UI objects directly, no synchronization is necessary within the UI library when the UI thread accesses UI elements. The only required synchronization in the UI library is on the shared queue where background threads enqueue tasks to run on the UI thread.
- Simple dynamic enforcement: Any library method that is intended to run only on the UI thread can contain an assertion that the current thread is the UI thread.

These advantages for the library implementor become sources of confusion and mistakes for client developers. Each method may be intended to run on the UI thread (and may therefore access UI elements) or may be intended to run on another, background, thread (and therefore must not access UI elements). Client developers must know at all times which thread(s) a given block of code might execute on. In cases where a given type or routine may be used sometimes for background thread work and sometimes for UI work, maintaining this distinction becomes even more difficult. There are alternative designs for GUI frameworks that alleviate some of this confusion, but they are undesirable for other reasons explained in Section 3.5.

The key insight of our work is that a simple type-and-effect system can be applied to clients of UI frameworks to detect all UI thread-access errors statically. There is a one-time burden of indicating which UI framework methods can be called only on the UI thread, but this burden is tractable. Annotations in client code are kept small thanks to a judicious use of default annotations and simple effect polymorphism.

We present a sound static polymorphic effect system for verifying the absence of (and as a byproduct, finding) UI thread access errors. Specifically, we:

- Present a concise formal model of our effect system, λ_{UI} . (Section 2)
- Describe an implementation $\mathsf{Java}_{\mathsf{UI}}$ for the Java programming language, including effect-polymorphic types, that requires no source modifications to UI libraries or clients beyond Java annotations for type qualifiers and effects. (Section 3)
- Evaluate $\mathsf{Java}_{\mathsf{UI}}$ by annotating 8 UI programs and Eclipse plugins totalling over 140KLOC. Our experiments confirm bugs found by unsound previous work [2],

find an additional bug, and verify the absence of such bugs in several large programs. (Section 4)
- Identify coding and design patterns that cause problems for our effect system and probably for other effect systems as well, and discuss possible solutions. (Section 4.4)

Our experience identifying UI errors in large existing code bases is likely to prove useful for designing other program analyses that may have nothing to do with UI code. Applying a static type-and-effect system to a large existing code base requires a design that balances expressiveness with low annotation burden. In particular, we found that while effect polymorphism was important, only a limited form (one type-level variable per class) is needed to verify almost all code. However, some programming idioms that do occur in practice are likely to remain beyond the reach of existing static analyses. Overall, we believe our work can pave the way for practical static analyses that enforce heretofore unchecked usage requirements of modern object-oriented frameworks.

Java_{UI} and our annotated subject programs are publicly available at:

`http://github.com/csgordon/javaui`

2 Core Language λ_{UI}

The basis for our Java type system is λ_{UI}, a formal model for a multithreaded lambda calculus with a distinguished UI thread. Figure 1 gives the syntax and static semantics for the core language. The language includes two constructs for running an expression on another thread: $\mathsf{spawn}\{e\}$ spawns a new non-UI thread that executes the expression e, while $\mathsf{asyncUI}\{e\}$ enqueues e to be run (eventually) on the UI thread. There are also two kinds of references: $\mathsf{ref}_{\mathsf{safe}}\ e$ creates a standard reference, while $\mathsf{ref}_{\mathsf{ui}}\ e$ creates a reference that may be dereferenced only on the UI thread. Other constructs are mostly standard (dereference, application, function abstraction, assignment, natural numbers, and a unit element) though the lambda construct includes not only an argument and body, but an effect bound ξ on the body's behavior.

Effects ξ include ui for the effect of an expression that must run on the UI thread, and safe for the effect of an expression that may execute on any thread. Types include natural numbers, unit, the standard effectful function type, and reference types with an additional parameter describing the effect of dereferencing that reference.

Figure 1 also gives the static typing rules for λ_{UI} . The form $\Gamma \vdash e : \tau; \xi$ can be read as: given a type environment Γ, the expression e evaluates to a value of type τ, causing effects at most ξ. Most of the rules are fairly standard modulo the distinctions between $\mathsf{spawn}\{e\}$ and $\mathsf{asyncUI}\{e\}$ and between $\mathsf{ref}_{\mathsf{safe}}\ e$ and $\mathsf{ref}_{\mathsf{ui}}\ e$. The rules also assume a least-upper-bound operator on effects ($\sqcup$) for combining multiple effects, and the rules include T-SubEff for permitting safe bodies in ui functions. Richer subtyping/subeffecting, for example full subtyping on function types, would be straightforward to add.

The operational semantics in Figure 2 are mostly standard, with the exception that the distinguished UI thread has a FIFO queue of expressions to execute. After reducing its current expression to a value it dequeues the next expression if available. The expression reduction relation is labeled with the effect of a given action, and the background

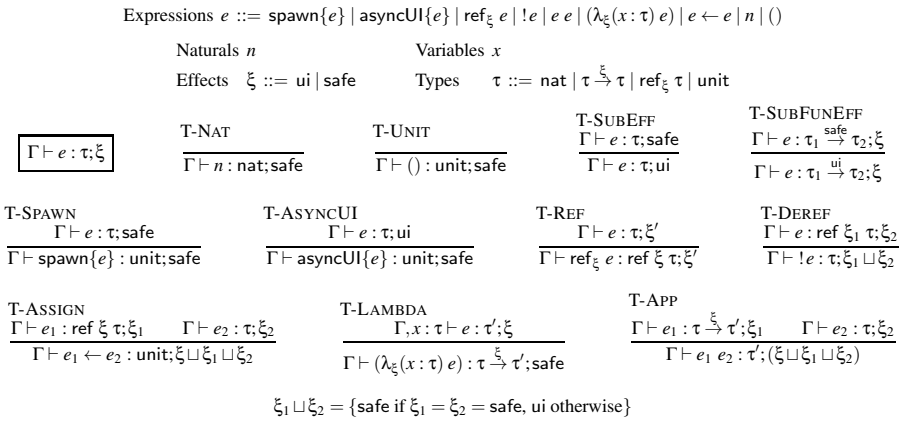


Fig. 1. λ_{UI} syntax for terms and types, and monomorphic type and effect system

threads get "stuck" if the next reduction would have the UI effect.[1] Typing for runtime program states is given in Figure 3.

The type system in Figure 1 is sound with respect to the semantics we define in Figure 2 (using the runtime and state typing from Figure 3) and is expressive enough to demonstrate soundness of the effect-related subtyping in the source language $\mathsf{Java}_{\mathsf{UI}}$. λ_{UI} models the ability to statically confine actions to a distinguished thread. Effect-polymorphism is not modeled, but mostly orthogonal: it would require a handful of additional checks at polymorphic instantiation, and an extended subeffecting relation ($\mathsf{safe} \sqsubset \mathsf{polyui} \sqsubset \mathsf{ui}$). Full proofs via syntactic type soundness [3] are available in our technical report [4]. We state the additional notation, judgments, and main lemmas here:

- Σ for standard-style heap type.
- $\Sigma \vdash H$ for typing the heap H.
- $\Sigma;\Gamma \vdash e : \tau;\xi$ as the expression typing judgment extended for heap typing.
- $H, e \rightarrow_\xi H', e', O$ for small-step expression reduction. The effect ξ is the runtime-observed effect of the reduction; e.g., dereferencing a ui reference has the runtime effect ui. O is an optional effect-expression pair for reductions that spawn new threads; the effect indicates whether a new background thread is spawned or a new UI task is enqueued for the UI thread.
- $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle$ for machine typing: the heap type, a vector of expression types for the UI thread's pending (and single active) expressions, and a vector of expression types for background threads. The maximal effects for each expression are implicit; they depend on which thread the expression is for: UI expressions may type with effect ui, while background threads must type with the effect safe).
- $\langle H, \overline{e_u}, \overline{e} \rangle$ for machine states: heap, pending UI expressions, and background threads, as with machine state typing.
- $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ for typing a machine state.

[1] This requires labeling heap cells with the effect of dereferencing the cell. These labels are used only for proving soundness and need not exist in an implementation.

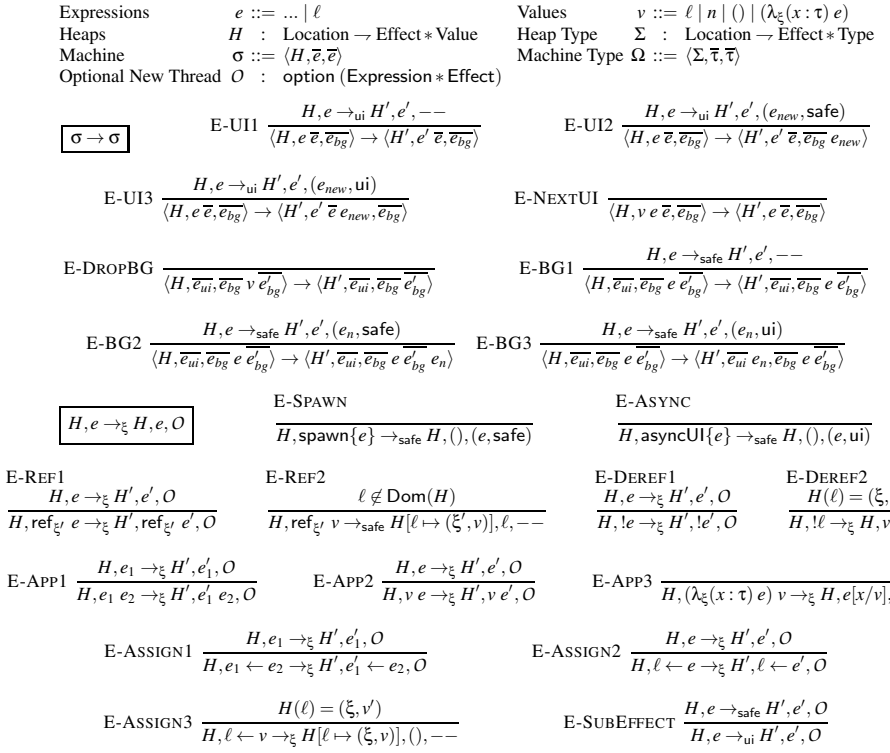


Fig. 2. λ_{UI} runtime expression syntax and operational semantics

– $\langle H, \overline{e_u}, \overline{e} \rangle \to \langle H', \overline{e'_u}, \overline{e'} \rangle$ as machine reduction, nondeterministically selecting either the first UI expression or an arbitrary background thread expression to reduce with the heap.

We prove soundness using syntactic type soundness [3]. First, we prove single-threaded type soundness. Contingent on that result, we prove soundness for all threads: all operations with the UI effect execute on the UI thread.

Lemma 1 (Expression Progress). *If* $\Sigma \vdash H$ *and* $\Sigma; \Gamma \vdash e : \tau; \xi$ *then either e is a value, or there exists some* H', e', *and O such that* $H, e \to_\xi H', e', O$.

Lemma 2 (Expression Preservation). *If* $\Sigma; \Gamma \vdash e : \tau; \xi$, $\Sigma \vdash H$ *and* $H, e \to_\xi H', e', O$, *then there exists* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \vdash H'$, $\Sigma'; \Gamma \vdash e' : \tau; \xi$, *and if* $O = (e'', \xi')$ *then there also exists* τ_n *such that* $\Sigma; \Gamma \vdash e'' : \tau_n; \xi'$.

Corollary 1 (Expression Type Soundness). *If* $\Sigma; \Gamma \vdash e : \tau; \xi$, *and* $\Sigma \vdash H$, *then e is a value or there exists* $\Sigma' \supseteq \Sigma$, e', H', *and O such that* $\Sigma' \vdash H'$, *and* $H, e \to H', e', O$, *and* $\Sigma'; \Gamma \vdash e' : \tau; \xi$ *and if* $O = (e'', \xi')$ *then there exists* τ_n *such that* $\Sigma'; \emptyset \vdash e'' : \tau_n; \xi'$.

Lemma 3 (Machine Progress). *If* $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ *for non-empty* e_u, *then either* $\overline{e_u} = v :: []$ *and* $\overline{e} = \emptyset$ *or there exists* $H', \overline{e_u}', \overline{e}'$ *such that* $\langle H, \overline{e_u}, \overline{e} \rangle \to \langle H', \overline{e_u}', \overline{e}' \rangle$.

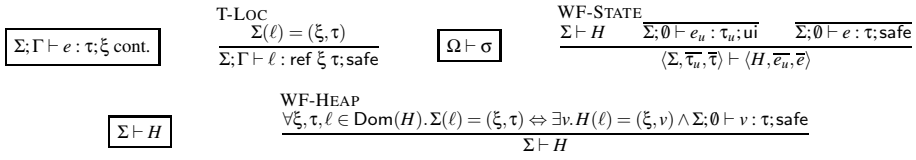


Fig. 3. λ_{UI} program and runtime typing, beyond extending the source typing with the additional Σ

Lemma 4 (Machine Preservation). *If* $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ *and* $\langle H, \overline{e_u}, \overline{e} \rangle \rightarrow \langle H', \overline{e_u}', \overline{e}' \rangle$, *then there exists* $\Sigma', \overline{\tau_u}', \overline{\tau}'$ *such that* $\Sigma' \supseteq \Sigma$ *and* $\langle \Sigma', \overline{\tau_u}', \overline{\tau}' \rangle \vdash \langle H', \overline{e_u}', \overline{e}' \rangle$

Corollary 2 (Machine Type Soundness). *If* $\langle \Sigma, \overline{\tau_u}, \overline{\tau} \rangle \vdash \langle H, \overline{e_u}, \overline{e} \rangle$ *for non-empty* e_u, *then either* $\overline{e_u} = v :: []$ *and* $\overline{e} = \emptyset$ *or there exists* $\Sigma', H', \overline{\tau_u}', \overline{e_u}', \overline{\tau}', \overline{e}'$ *such that* $\Sigma' \supseteq \Sigma$ *and* $\langle \Sigma', \overline{\tau_u}', \overline{\tau}' \rangle \vdash \langle H', \overline{e_u}', \overline{e}' \rangle$ *and* $\langle H, \overline{e_u}, \overline{e} \rangle \rightarrow \langle H', \overline{e_u}', \overline{e}' \rangle$.

3 Java$_{\mathsf{UI}}$: Extending λ_{UI} to Java

Java$_{\mathsf{UI}}$ soundly prevents inappropriate access to UI objects by background threads. Java$_{\mathsf{UI}}$ extends Java with type qualifier annotations and method effect annotations to indicate which code should run on the UI thread and which code should not directly access UI objects. From the Java developer's perspective, Java$_{\mathsf{UI}}$ prevents invalid thread access errors, which occur when a background thread calls UI-only methods (such as `JLabel.setText()`) that may only be called from the UI thread. Java$_{\mathsf{UI}}$ handles the full Java language, including sound handling of inheritance and effect-polymorphic types. A major design goal for Java$_{\mathsf{UI}}$ was to avoid changes to UI library code. Specifically, we did not modify the implementation or underlying Java type signature of any library.

We implemented our qualifier and effect system on top of the Checker Framework [5,6], which is a framework for implementing Java 8 type annotation processors, providing support for, among other things, AST and type manipulation, and specifying library annotations separately from compiled libraries themselves. The Java$_{\mathsf{UI}}$ syntax is expressed via Java 8's type annotations. Type annotations are propagated to bytecode, but have no runtime effect, thus maintaining binary API compatibility between Java$_{\mathsf{UI}}$ and Java code (developers use the real UI libraries, not one-off versions coupled to Java$_{\mathsf{UI}}$). The implementation consists of two main components: a core checker for the effect system, and a sizable annotation of some standard Java UI framework libraries.

3.1 Java$_{\mathsf{UI}}$ Basics

There are two method annotations that indicate whether or not code may access UI objects (call methods on UI objects):

– `@UIEffect` annotates a method that may call methods of UI objects (directly or indirectly), and must therefore run on the UI thread.

Table 1. Java$_{UI}$ annotations

Role	Annotation	Target	Purpose
Effects	@SafeEffect	Method	Marks a method as safe to run on any thread (default)
	@UIEffect	Method	Marks a method as callable only on the UI thread
	@PolyUIEffect	Method	Marks a method whose effect is polymorphic over the receiver type's effect parameter
Defaults	@UIType	Type Decl.	Changes the default method effect for a type's methods to @UIEffect
	@UIPackage	Package	Changes the default method effect for all methods in a package to @UIEffect
	@SafeType	Type Decl.	Changes the default method effect for a type's methods to @SafeEffect (useful inside a @UIPackage package)
Polymorphism	@PolyUIType	Type Decl.	Marks an effect-polymorphic type (which takes exactly one effect parameter)
Instantiating Polymorphism	@Safe	Type Use	Instantiates an effect-polymorphic type with the @SafeEffect effect (also used for monomorphic types, considered subtypes of @Safe `Object`)
	@UI	Type Use	Instantiates an effect-polymorphic type with the @UIEffect effect
	@PolyUI	Type Use	Instantiates an effect-polymorphic type with the @PolyUIEffect effect (the effect parameter of the enclosing type)

– @SafeEffect annotates a method that does *not* call methods of UI objects, and may therefore run on any thread.

@UIEffect corresponds to λ_{ui} from the λ_{UI} core language: it annotates a method as having a (potentially) UI-effectful body. Similarly, @SafeEffect corresponds to λ_{safe}.

The other annotations in Table 1 are all about changing the default effect (@UIType, @SafeType, @UIPackage) or handling polymorphism (@PolyUIEffect, @PolyUI, @Safe, @UI, @PolyUIType).

A @SafeEffect method (the default) is not permitted to call @UIEffect methods. Only @UIEffect methods (which have the UI effect) are permitted to call other @UIEffect methods. With few exceptions (e.g., `syncExec()`, which background threads may call to enqueue an action to run on the UI thread), methods of UI elements are annotated @UIEffect, and that is how improper calls to UI methods are prevented:

```
public @SafeEffect void doSomething(JLabel l) {
    l.setText(...); // ERROR: setText() has the UI effect
}
```

The other method effect annotation @UIEffect annotates a method as able to call all @UIEffect methods, and only other methods with the UI effect may call it.

```
public @UIEffect void doSomethingUI(JLabel l) {
    l.setText(...); // OK: setText() and this method have UI effect
}
public @SafeEffect void doSomethingSafe(JLabel l) {
    doSomethingUI(l); // ERROR: doSomethingUI() has the UI effect
}
```

The safe effect @SafeEffect is a subeffect of the UI effect @UIEffect. So a @SafeEffect method may override a @UIEffect method, but a @UIEffect method cannot override a @SafeEffect method:

```
public class Super { public @SafeEffect void doSomethingSafe(){ ... }}
public class Sub extends Super {
    public @UIEffect void doSomethingSafe(){ ... }}
           //^ ERROR: invalid effect override
```

Controlling Defaults. Especially for subclasses of UI elements, it could be tedious to write @UIEffect on every method of a class. Three additional annotations locally change the default method effect:

- @UIType: A class (or interface) declaration annotation that makes all methods of that class, including constructors, default to the UI effect.
- @UIPackage: A package annotation that makes all types in that package behave as if they were annotated @UIType. Subpackages must be separately annotated; @UIPackage is not recursive.
- @SafeType: Like @UIType, but changes the default to @SafeEffect (useful inside @UIPackage packages).

In all three cases, individual methods may be annotated @SafeEffect or @UIEffect . This is how we have annotated SWT, Swing, and JFace: we annotated each package @UIPackage, then annotated the few safe methods (such as `repaint()`) as @SafeEffect.

3.2 Effect-Polymorphic Types

A single class may be used sometimes for UI-effectful work, and other times for work safe to run on any thread. @SafeEffect and @UIEffect do not handle this common use case. In particular, consider Java's `java.lang.Runnable` interface:

```
public interface Runnable {
    void run();
}
```

The `Runnable` interface is used to encapsulate both code that must run on the UI thread (which is passed to the `syncExec()` and `asyncExec()` methods in UI libraries), and also code that should not run on the UI thread, such as the code passed to various general dispatch queues, including thread pools.

Java$_{UI}$ provides three type qualifiers [5,6,7], each corresponding to instantiating an effect-polymorphic type with a specific effect:

- @Safe: A qualifier to instantiate a type with the @SafeEffect effect.
- @UI: A qualifier to instantiate a type with the @UIEffect effect.
- @PolyUI: A qualifier to instantiate a type with the @PolyUIEffect effect (when used inside an effect-polymorphic class).

as well as a type declaration annotation @PolyUIType that annotates a class or interface as effect-polymorphic.[2]

The final declaration for the `Runnable` interface is:

```
@PolyUIType public interface Runnable {
    @PolyUIEffect void run();
}
```

This declares `Runnable` as being a class polymorphic over one effect, and the effect of the `run()` method is that effect. Our type system implicitly adds a receiver qualifier of @PolyUI so the body can be checked for any instantiation.

Given an instance of a `Runnable`, the effect of calling the `run()` method depends on the qualifier of the `Runnable`. For example:

```
@Safe Runnable s = ...;
s.run(); // has the safe effect
@UI Runnable u = ...;
u.run(); // has the UI effect
@PolyUI Runnable p = ...;
p.run(); // has a polymorphic effect
```

Assuming the last line appears inside an effect-polymorphic type, its effect will be whatever effect the type is instantiated with. Note that the @Safe annotation on `s` is not necessary: the default qualifier for any type use is @Safe if none is explicitly written. Since most code does not interact with the UI, this means that most code requires no explicit annotations.

Effect-Monomorphic Subtypes of Effect-Polymorphic Supertypes. Deriving a concrete subtype from an effect-polymorphic supertype is as simple as writing the appropriate qualifier on the supertype:

```
public class SafeRunnable implements @Safe Runnable {
    @SafeEffect void run() {
        // any effect other than safe causes a type error here
    }
}
```

Again, note that @SafeEffect is the default effect for unannotated methods, so the use of @SafeEffect here is not strictly necessary. Inside the body of `run()`, this will have type @Safe `SafeRunnable`. In this case, the @Safe could be omitted from the `implements` clause due to defaults, but a class that implements @UI Runnable would require the explicit qualifier.

[2] Java$_{UI}$ uses different annotations for monomorphic effects (@SafeEffect) and for instantiating polymorphic effects (@Safe) because of a parsing ambiguity in Java 8 where method annotations and return type annotations occupy the same source locations.

It is also possible to derive from a polymorphic type *without* instantiating, by simply declaring a polymorphic type that derives from a polymorphic instantiation of the supertype:

```
@PolyUIType public interface StoppableRunnable implements @PolyUI Runnable {
    @PolyUIEffect void run();
    @PolyUIEffect void stop();
}
```

Concrete or abstract classes may be effect-polymorphic. The body of an effect-polymorphic method is limited to calling safe methods and other methods with their same effect (e.g., other effect-polymorphic methods of the same instance, which will have the same effect as the executing method).

It is not permitted to derive a polymorphic type from a non-polymorphic type (see Section 3.4 for why). Therefore, `Object` is declared as effect-polymorphic.

Qualifier Subtyping and Subeffecting. In addition to nominal subtyping (e.g., @Safe `SafeRunnable` above is a subtype of @Safe `Runnable`), Java_{UI} also permits qualifier subtyping, which reflects effect subtyping ("subeffecting"). For example, it may be useful to pass a @Safe `Runnable` where a @UI `Runnable` is expected: any `run()` implementation with the safe effect is certainly safe to execute where a UI effect is allowed. Similarly, a @Safe `Runnable` can be passed in place of a @PolyUI `Runnable`, which can be passed in place of a @UI `Runnable` since both subtyping relations are sound for any instantiation of @PolyUI.

Anonymous Inner Classes. Many programs use anonymous inner classes to pass "closures" to be run on the UI thread. Qualifiers can be written on the type name. Thus code such as the following is valid (type-correct) Java_{UI} code:

```
asyncExec(new @UI Runnable() { public @UIEffect void run(){/* UI stuff */}});
```

If an effect-monomorphic supertype declares @UIEffect methods, no annotation is needed on the anonymous inner class, and all overriding methods default to the effect declared in the parent type, without additional annotation.

3.3 Annotating UI Libraries

To update UI elements, non-UI code uses special methods provided by the UI library that run code on the UI thread: the Java equivalent of λ_{UI}'s $\text{asyncUI}\{e\}$ construct. Real UI libraries have both synchronous and asynchronous versions:

- For Swing, these special functions are methods of `javax.swing.SwingUtilities`:
  - `static void invokeAndWait(Runnable doRun);`
  - `static void invokeLater(Runnable doRun);`
- For SWT, these special functions are methods of the class `org.eclipse.swt.widgets.Display`:
  - `static void syncExec(Runnable runnable);`
  - `static void asyncExec(Runnable runnable);`

Other UI libraries have analogous functionality. We annotated each of these library methods as @SafeEffect (safe to call on any thread) and accepting a @UI instance of the `Runnable` interface (allowing UI-effectful code in the `Runnable`). This is comparable to λ_{UI}'s T-AsyncUI in Figure 1. Our library annotations use the Checker Framework's *stub file* support [6,5] for stating trusted annotations for code in pre-compiled JAR files. We did not check the internals of GUI libraries, which would require dependent effects (Section 3.4).

3.4 Limitations

There are a few theoretical limits to this effect system. In our evaluation (Section 4), these did not cause problems.

One Effect Parameter Per Type. Java$_{UI}$ cannot describe the moral equivalent of

```
class TwoEffectParams<E1 extends Effect, E2 extends Effect> {
  @HasEffect(E1) public void m1() { ... }
  @HasEffect(E2) public void m2() { ... }
}
```

In our evaluation on over 140,000 lines of code (Section 4), this was never an issue. We found that effect-polymorphic types are typically limited to interfaces that are used essentially as closures. They are used for either safe code or UI code, rarely a mix. This restriction also gives us the benefit of allowing very simple qualifier annotations for instantiated effect-polymorphic types. Supporting multiple parameters would require a variable-arity qualifier for instantiating effects, and introduce naming of effect parameters. (We have found one instance of a *static method* requiring multiple effect parameters: `BusyIndicator.showWhile()`, discussed in Section 4.4.)

Polymorphic Types May not Extend Monomorphic Types. Java$_{UI}$ does not permit, for example, declaring a subtype `PolyUIRunnable` of `UIRunnable` that takes an effect parameter, because it further complicates subtyping. It is possible in theory to support this, but we have not found it necessary. To do so, effect instantiations of the effect-polymorphic subclass would instantiate only the *new* polymorphic methods of the subclass (polymorphic methods inherited from further up the hierarchy and instantiated by a monomorphic supertype may not be incompatibly overridden). Subtyping would then become more complex, as the qualifier of a reference could alternate almost arbitrarily during subtyping depending on the path through the subtype hierarchy.

Splitting the class hierarchy. Because an effect-polymorphic type may not inherit from a monomorphic type, this forces the inheritance hierarchy into three partitions: @UI types, @Safe types, and @PolyUI types (`Object` is declared as @PolyUI, making the root of the type hierarchy @UI `Object`). All may freely reference each other, but it does impose some restrictions on code reuse. This was not an issue in our evaluation. Some classes implement multiple interfaces that each dictate methods with different effects (e.g., a listener for a UI event and a listener for background events, each handler having a different effect; Eclipse's `UIJob` interface has methods of both effects), but we found no types implementing multiple polymorphic interfaces using different instantiations.

No effect-polymorphic field types. We do not allow effect-polymorphic (@PolyUI) fields. This avoids reference subtyping problems. Solutions exist (e.g., variance annotations [8]) but we have not found them necessary. Note however that we do inherit Java's unsoundness from covariant array subtyping, though we encountered no arrays of any effect-polymorphic types (or any @UI elements) during evaluation.

Cannot check UI library internals. The effect system currently is not powerful enough to check the internals of a UI library, mainly because it lacks the dependent effects required to reason about the different effects in separate branches of dynamic thread checks. This means for example the effect system cannot verify the internals of safe UI methods like `repaint()`, which are typically implemented with code like:

```
if (runningOnUIThread()) { /*direct UI access*/ } else { /*use syncExec()*/ }
```

3.5 Alternatives

There are four possible approaches to handling UI threading errors: unchecked exceptions (the approach used by most GUI libraries), a sound static type and effect system, a Java checked exception (a special case of effect systems), and making every UI method internally call `syncExec()` if called from a background thread. The unchecked exception is undesirable for reasons described in the introduction: the resulting bugs are difficult to diagnose and fix. We propose a sound static type system independent from Java checked exceptions, and most of this paper explores that option. This section focuses on the two remaining options, and why our approach is different from existing concurrency analyses.

Why not make the thread access exceptions checked? Java's checked exceptions are another sound static effect system that could prevent these bugs. But there are reasons to use a separate effect system rather than simply making the thread access error exception a checked (rather than the current unchecked) exception:

- Polymorphism: Certain types, such as `Runnable`, are used for both UI thread code and non-UI code — such types are polymorphic over an effect. Java does not support types that are polymorphic over whether or not exceptions are thrown. We aim to minimize changes to existing code and to avoid code duplication.
- Reducing annotation burden: For common source code structures, such as putting most UI code in a separate package, a programmer can switch the default effect for whole types or packages at a time. Java provides no way to indicate that all methods in a type or package throw a certain checked exception. Java_{UI} provides such shorthands.
- Backwards Compatibility: New checked exceptions breaks compilation for existing code. This is also the reason we do not leverage Java's generics support for our effect-polymorphic types.
- Catching such exceptions would almost always be a bug.

Why not have every UI method automatically use `syncExec()` *if run on a background thread?* This solution masks atomicity errors on UI elements. A background thread may call multiple UI methods — for example, to update a label and title bar together. Different background threads could interleave non-deterministically in this approach, creating inconsistencies in the user interface. Additionally, these atomicity bugs would hurt performance by increasing contention on the shared queue of messages from background threads due to the increased thread communication.

Why not use an existing concurrency analysis? Our effect system is different from prior type and effect systems for concurrency. Our goal is to constrain some actions to a specific distinguished thread, which is not a traditionally-studied concurrency safety property (as opposed to data races, deadlocks, and atomicity or ordering violations). In particular, this effect system permits most concurrency errors! This is by design, because preventing better-known concurrency errors is neither necessary nor sufficient to eliminate UI thread access errors, and allows the effect system design to focus on the exact error of interest. Data races on model structures are not UI errors. Because UI libraries dictate no synchronization other than the use of `syncExec()` and `asyncExec()` (and equivalents in other frameworks), deadlocks are not UI errors. It is also possible for a program to have a UI error without having any traditional concurrency bugs. Java$_{UI}$ only guarantees the UI library's assumption that all UI widget updates run uninterrupted (by other UI updates) in the same thread. In general, other static or dynamic concurrency analyses would complement Java$_{UI}$'s benefits, but the systems would not interact.

4 Evaluation

We evaluated the effectiveness of our design on 8 programs with substantial user interface components (Table 2). 4 of these programs were evaluated in prior work [2]. The others were the first 4 UI-heavy Eclipse plugins we could get to compile out of the 50 most-installed[3] (as of May 2012).

We wrote trusted annotations for major user interface libraries (Swing, SWT, and an SWT extension called JFace), annotated the programs, and categorized the resulting type-checking warnings. Where false warnings were issued due to the type system being conservative, we describe whether there are natural extensions to the type system that could handle those use cases.

4.1 Annotation Approach

Trusted Library Annotations. We conservatively annotated the UI libraries used by subject programs before annotating the programs themselves. Swing contains 1714 classes, SWT contains 708 classes, and JFace 537 classes. We erred on the side of giving too many methods the UI effect, and we adjusted our annotations later if we found them to be overly conservative. We intermingled revisions to library annotations with subject program annotation, guided by the compiler warnings (type errors). We examined the

[3] `http://marketplace.eclipse.org/metrics/installs/last30days`

Table 2. Subject programs. Pre-annotation LOC are calculated by `sloccount` [9]. UI LOC is the LOC for the main top-level package containing most of the application's UI code; other parts of a project may have received some annotation (for example, one method in a model might be executed asynchronously to trigger a UI update), and some projects were not well-separated at the package level.

Program	LOC	UI LOC	Classes	Methods
EclipseRunner	3101	3101	48	354
HudsonEclipse	11077	11077	74	649
S3dropbox	2353	1732	42	224
SudokuSolver	3555	3555	10	62
Eclipse Color Theme	1513	1193	48	215
LogViewer	5627	5627	117	644
JVMMonitor	31147	17657	517	2766
Subclipse	83481	53907	539	4480

documentation, and in some cases the source, for every library method that caused a warning. When appropriate, we annotated library methods as @SafeEffect, annotated polymorphic types and effects for some interfaces, and changed some UI methods to accept @UI instantiations of effect-polymorphic types. The annotated library surface is quite large: we annotated 160 library packages as @UIPackage, as well as specifying non-default effects for several dozen classes (8 effect-polymorphic).

Our results are sound up to the correctness of our library annotations and the type checker itself. We can only claim the UI framework annotations to be as accurate as our reading of documentation and source. A few dozen times we annotated "getter" methods that returned a field value as safe when it was not perfectly clear they were *intended* as safe. There are three primary sources of potential unsoundness in library annotations:

1. Incorrectly annotating a method that does perform some UI effect as safe.
2. Incorrectly annotating a method that requires a safe variant of a polymorphic type as accepting a UI variant.
3. Incorrectly annotating a callback method invoked by a library as @UIEffect .

To mitigate the first source, we began the process by annotating every UI-related package and subpackage we could find as @UIPackage. Java_{UI} mitigates the second by the fact that unspecified polymorphic variants default to safe variants. We addressed the third by reading the documentation on the several ways the UI frameworks start background threads, and annotating the relevant classes correctly early on.

Annotating Subject Programs. To annotate each subject program, we worked through the files with the most warnings first. We frequently annotated a class @UIType if most methods in the class produced warnings; otherwise we annotated individual methods with warnings @UIEffect . For files with fewer warnings, we determined by manual code inspection and perusing UI library documentation whether some methods came from an interface with UI-effectful methods, annotating them @UIEffect if needed.

In an effort to make the final warning count more closely match the number of possible conceptual mistakes, when the body of a method that must be safe (due to its

use or inheritance) contained one basic block calling multiple @UIEffect methods (e.g. `myJLabel.setText(myJLabel.getText()+"...")`), we annotated the method body @UIEffect, taking 1 warning over possibly multiple warnings about parts of the method body. We believe this makes the final warning counts correspond better to conceptual errors. Multiple UI method calls in a safe context likely reflects 1 missing `asyncExec()` (a developer misunderstanding the calling context for a method) or 1 misunderstanding on our part of the contexts in which a method is called, not multiple bugs or misunderstandings. If multiple separated (different control flow paths) basic blocks called @UIEffect methods, we left the method annotation as @SafeEffect.

We made no effort during annotation to optimize annotation counts (the counts we do have may include annotations that could be removed). We simply worked through the projects as any developer might.

We identified several patterns in library uses that cause imprecision; we discuss those in Section 4.4.

Distinguishing warnings that correspond to program bugs, incorrect (or inadequate) annotations, tool bugs, or false positives requires understanding two things: the semantics of Java$_{UI}$'s effect system, and the intended threading design of the program (which code runs on which thread). The user has detected a program bug or a bad annotation when JavaUI indicates that the program annotations are not consistent with the actual program behavior. Finding the root cause may require the user to map the warning to a call path from a safe method to a UI method. This is similar to other context- and flow-sensitive static analyses, and when the user does not understand the program, iteratively running JavaUI can help. The user can recognize a false positive by understanding what is expressible in JavaUI. A reasonable rule of thumb is that if a warning could be suppressed by conditioning an effect by a value or method call whose result depends on the thread it runs on, it is likely a false positive. The user can recognize a tool bug in the same way, by understanding JavaUI's simple rules.

4.2 Study Results: Bugs and Annotations

Annotating these projects taught us a lot about interesting idioms adopted at scale, and about the limitations of our current effect system. The annotation results appear in Table 3, including the final warning counts and classifications, and Table 4 gives the number of each annotation used for each project. The first four projects each took under an hour[4] to annotate, by the process described in Section 4.1. Eclipse Color Theme took only 8 minutes to annotate, and required only 4 annotations. The effort required for these five projects was low even though we had never seen the code before starting annotation. The other projects (LogViewer, JVMMonitor, and Subclipse) were substantially larger and more complex, and they required substantially more effort to annotate.

Overall we found all known bugs in the subject programs (only the first four were known to have UI threading related errors [2]), plus one new UI threading defect, and one defect in unreachable (dead) code.

[4] We lack precise timing information because each annotation was interleaved with fixing Java$_{UI}$ implementation bugs.

Table 3. Java$_{UI}$ warnings (type errors). Java$_{UI}$ finds all bugs found by Zhang's technique [2], plus one additional bug. The table indicates UI threading defects, non-exploitable code defects found because of annotation, definite false positives, and a separate category for other reports, which includes reports we could not definitively label as defects or false positives, as well as other warnings such as poor interactions between plugins' Java 1.4-style code and the 1.7-based compiler and JDK we used.

	Zhang et al. [2]				Defects			
Program	Warnings	Defects	Time to Annotate	Warnings	UI	Other	False Pos.	Other
EclipseRunner	6	1	<1hr	1	1	0	0	0
HudsonEclipse	3	3*	<1hr	13**	3	0	2	0
S3dropbox	1	1	<1hr	2	2	0	0	0
SudokuSolver	2	2	<1hr	2	2	0	0	0
Eclipse Color Theme	0	0	8m	0	0	0	0	0
LogViewer	0	0	3h50m	1	0	0	1	0
JVMMonitor	7	0	6h45m	9	0	0	9	0
Subclipse	24	0	17h20m	19	0	1	13	5

*Zhang et al. report 1 bug, but its repair requires adding `syncExec()` in 3 locations, so we consider it 3 bugs.
**Java$_{UI}$ found the same 3 bugs as Zhang et al.'s GUI Error Detector, each with 3–4 warnings due to compound statements.

Table 4. Annotation statistics for the 8 subject programs. @PolyUIEffect, @PolyUIType, and @PolyUI were not used in the subject programs themselves — only in annotating the UI libraries.

Program	@UIPackage	@UIType	@SafeType	@UIEffect	@SafeEffect	@UI	@Safe	Anno/KLOC
EclipseRunner	0	26	0	2	5	0	0	10.6
HudsonEclipse	0	17	0	9	4	14	0	3.9
S3dropbox	0	30	0	4	2	14	0	21.2
SudokuSolver	0	2	0	18	1	9	0	8.4
Eclipse Color Theme	0	1	0	3	0	0	0	2.6
LogViewer	0	53	0	5	23	15	1	17.2
JVMMonitor	0	129	0	12	47	29	0	6.9
Subclipse	17	126	60	102	128	138	0	6.8

Table 3 includes the error counts for Zhang et al.'s unsound GUI Error Detector [2] when run on the same program versions. We found all bugs they identified, as well as 1 new bug in S3dropbox. The S3dropbox developer has confirmed the bug, though he does not plan to fix it because it does not crash the program (Swing does not check the current thread in every UI method, allowing some races on UI objects). If a user drops a file using drag-and-drop onto the UI, the interface sometimes forks a background thread that then calls back into UI code. GUI Error Detector misses this bug because of an unsoundness in WALA [10], which GUI Error Detector uses for call graph extraction. For scalability, by default WALA does not analyze certain libraries, including Swing. GUI Error Detector uses WALA's default settings, so the drag-and-drop handler appears (to the tool) to be unreachable. Call graph construction precision is also a bottleneck for GUI Error Detector on large programs: Subclipse analysis required a less precise control flow analysis to finish (0-CFA, others used 1-CFA).

The dead code defect we found was a UI-effectful implementation of a safe interface in Subclipse. The type with the invalid override was never used at that interface type (removing the `implements` clause fixes the warning) and the method was never called. We consider this a defect, though it is not exploitable.

In Table 3 the number of final warnings exceeding the number of bugs found does not necessarily indicate false positives: our type system issues a warning for every type-incorrect *expression* that could correspond to a thread access error. So a single line with a composite UI expression (e.g., a UI method call with UI expressions as arguments) in a non-UI context would (correctly) produce multiple compiler warnings. Each subexpression may have an individual work-around that does not require adding an `(a)syncExec()`. We consider a warning to be a false positive only if the target expression's execution in context would not improperly access UI elements from a non-UI thread.

Our sound type and effect system found no new UI threading errors in the larger projects, but found several bugs in the smaller projects. There are good reasons to expect this result. First, the first four projects were previously evaluated by Zhang et al. [2], so we knew we should find bugs in those projects (Zhang selected several of those subjects by finding SWT threading errors in bug databases). For the larger projects, we simply took the most recent release version of each Eclipse plugin. Second, the larger Eclipse plugins are all mature, heavily used projects, making it quite likely that any UI bugs would be found and fixed quickly (each has at least 3,000 installs total; Subclipse was installed 23,855 times between 11/19 and 12/19 2012 alone). We believe Java$_{UI}$ would have more benefits when used from the beginning of a project, and the relative prevalence of UI errors in the younger projects compared to the mature projects supports this theory.

After annotating these programs, we searched all four projects' issue trackers for threading-related issues. There were several bug reports for data races between models and views, and issues with several UI methods that behave differently on the UI thread than on other threads; the latter are not uncommon, because many JFace methods return one result (often null) on non-UI threads, but a different result on the UI thread. The latter could have been caught by our type system with custom library annotations for those methods.

The one report we found of a UI threading error [11] is triggered when a background thread calls into a native method, which then calls back into UI-effectful Java code. The call occurs as a result of a logic error in Subversion itself, and the bug was marked "WONTFIX" (the fix was a patch to Subversion). With a proper annotation on the native methods, our effect system would have issued a warning.

Non-annotation Changes. We made minor changes to the code beyond simply adding effect-related annotations (and the required `import` statements) for two reasons: when naming an anonymous inner class as a new subtype would fix effect (type) errors, and when converting Java 1.4-style code to use generics would remove warnings. The `Action` interface is generally used as a closure for @UIEffect work, but HudsonEclipse in one case made an anonymous inner class whose `run()` method was safe, stored it as an `Action` (whose `run()` is @UIEffect), and called the `run()` of the safe subclass explicitly in several safe contexts. Rather than making `Action` (and several supertypes) polymorphic, which seems to contradict the suggested uses in the documentation, we declared a subclass of `Action` that overrode `run()` as @SafeEffect, and stored the anonymous inner class as an instance of that (removing 3 warnings). In JVMMonitor, Java 1.4-style (no generics) use of Java Beans interacted poorly with the Checker Framework's promotion of an argument of type `Class` to `Class<? extends @Safe Object>`, which was passed as an argument to a Java Bean method accepting

Client code listener (callback) implementations:

```
class SafePropertyChangeListener extends PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent event) {
        if (event.getProperty().equals("stuff")) {
            // do @SafeEffect stuff
}}}
class UIPropertyChangeListener extends PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent event) {
        if (event.getProperty().equals("uistuff")) {
            // Call @UIEffect methods directly
}}}
```

On UI thread:

```
store.addPropertyChangeListener(new UIPropertyChangeListener());
```

On a background thread:

```
store.addPropertyChangeListener(new BackgroundPropertyChangeListener());
// Next line executes UIPropertyChangeListener's UI callback on BG thread
store.setValue("stuff", true);
```

Fig. 4. JFace global property store issues. Assume `store` is any expression that accesses a shared static JFace `PreferenceStore`; in Eclipse plugins, there is one such store initialized for every plugin. Listeners for property changes are registered with both possible effects, but all handlers will run on any thread that updates any property, making UI thread errors possible. As long as specific properties that actually cause @UIEffect methods to be called (in this case, `uistuff`) are only updated from the UI thread, no errors will occur, but this pattern is fragile.

a `Class<@Safe Object>`. We added a generic method parameter `T` and changed the argument to `Class<T>`, removing one warning. In Subclipse, we converted 3 `Vector` fields to `Vector<String>`, due to a similar problem with `Vector.copyInto`, removing 3 warnings.

We allowed these changes because they were minor, and because we expect they would be natural changes for a project interested in using our effect system to verify the absence of UI threading errors. Most of the remaining false positives could be fixed with more engineering work (e.g. splitting interfaces, splitting callback registration into safe and UI-effectful, etc.) but we judged such changes to be too invasive to give a clear picture of developer effort, and restricted ourselves to only these small changes.

4.3 False Positives and Other Reports

Our evaluation produced 30 false positives in over 140,000 lines of code (0.2 per 1000 lines of code). We consider this an acceptable false positive rate. The false positives fall into 5 general categories, including limitations of our type system and what we consider to be poor designs in the UI libraries and client programs.

Registering Callbacks of Both Effects. Four of the projects shared a common source of false positives: HudsonEclipse (1 false positive), LogViewer (1), JVMMonitor (5), and

Subclipse (1) each suffered imprecision from JFace's global property store. The plugin code adds a UI-effectful property change listener (a @UI instantiation of a polymorphic interface) to JFace's global property store. Listeners will fire on any thread that sets a property, so in general the listeners must be safe. However, in some projects all calls to setting properties are performed from within the UI thread, making this not a bug. A potential solution for this class of false positives is to change the library annotations for JFace's preference store to permit @UI handlers, and to annotate the property setters @UIEffect in a project-specific library annotation file used to override the global file. (The global file would follow documentation as much as possible.) In other cases, particular properties are only updated from the UI thread, and the UI-effectful handlers are guarded by condition checks that only pass for those UI properties, as in Figure 4.

JVMMonitor created its own specific instance of this same problem: the other 4 of its false positives are from a property store it creates for CPU model changes, where some properties are only updated from UI code, and the handlers have UI effects but run only for specific properties. We feel this property store design is faulty, which we elaborate on in Section 4.4.

Subtyping Limitations. 5 other false positives (1 in HudsonEclipse and 4 in Subclipse) are from a weakness in our subtyping relation. HudsonEclipse's subtyping false positive occurs because a @UI instantiation of a polymorphic type is not a subtype of @Safe `Object`. The subtyping false positives in Subclipse are from a combination of that with the requirement that generic parameters are subtypes of the default Object variant: the code uses a Java 1.4 style list, but `List<T>` is implicitly `List<T extends` @Safe `Object>` which makes some types unusable as type parameters. These uses would be enabled by making the upper bound on `List` (and `Iterator`) @UI `Object`, but doing so would force many additional annotations where an object was pulled out of a list or iterator as (implicitly safe) `Object`, so we opted for the lower annotation burden. This would not be an issue in properly generic code.

Interface Abuse. One false positive in Subclipse is an instance of *interface abuse*: subclassing a definitively safe supertype using @UIEffect overrides, then calling the @UIEffect overrides directly, only from UI contexts. We could have fixed this type error by making the entire hierarchy of the abusing class's superclasses effect-polymorphic (from documentation one superclass is clearly intended as safe), or by introducing a new type and copying code from parent classes (which is poor design for its own reasons).

Lack of Dependent Types/Effects. Subclipse had 7 additional definite false positives, most of which would require dependent effects (as in dependent types; an effect determined by a runtime result) to handle:

- 3 warnings resulting from lack of dependent effects (see Section 4.5).
- 3 warnings that were subject to some type of dynamic thread check, and would therefore have executed only on the UI thread
- 1 instance where our type system could not express the proper effect (it would require a combination of dependent effects with multiple method effect parameters and explicit least-upper-bound-of-parameters effects)

Other Reports. The remaining 5 Subclipse reports are about unsafe effects, but we cannot determine whether or not the application is using a safe interface as polymorphic (at a @UI variant) or if the JFace interface, documented as unrelated to interfaces or threads, should actually just allow UI effects.

4.4 Sources of Difficulty

Weak Documentation. The main source of difficulty in annotating all of our subject programs was understanding the design of the UI libraries, with respect to which methods must only be called in the UI thread, or were polymorphic. Once we understood the design, adding library annotations was easy. Remarkably, none of the UI libraries clearly and consistently documents the thread safety of all UI methods. Java_{UI}'s annotations are precise documentation, and are machine-checkable for client code.

AWT and Swing's documentation was concise and unambiguous: all methods except `invokeAndWait` and `invokeLater` must be called only from the UI thread. SWT claims the same about `syncExec()` and `asyncExec()`. Confusingly, there are some exceptions to this rule in SWT (classes `Color`, `Font`, and `Image`).

The prevailing wisdom about JFace is that most of JFace assumes it is running on the UI thread. But clients call much of JFace directly from non-UI threads, and the JFace documentation rarely specifies thread assumptions. Clients often interpret the lack of documentation as license to call: there are many methods of UI elements that documentation suggests are intended to be called only from the UI thread, but happen to be safe (such as getter methods) and are therefore often called by clients from contexts that must be safe.

Problematic Idioms. There are several idioms that cause problems for our type system. Most could be handled with richer polymorphism or dependent effects, but we believe most of these idioms are poor design. Rewriting the offending code is a better option, and Java_{UI}'s type system encourages this better design.

The most common source of false positives was JFace's global property store design. JFace often shares global property sets among all threads, and listeners can be registered for a callback in the event of a property changing. These listener callbacks will be executed on whichever thread updates a property, thus all properties callbacks should be @SafeEffect. However, some programs register @UIEffect callbacks, but avoid issues by updating the global property store (for any property) only from the UI thread, or updating the properties with UI-effectful handlers only from the UI thread. Two of these safe approaches can be handled using custom library annotations for individual projects: either callbacks can have the UI effect and properties may only be updated from the UI thread, or callbacks must be safe and the properties may be updated by any thread. The related case as seen with the CPU model change listeners in JVMMonitor could be handled with some dependent effects, annotating the listener update code with the properties whose handlers may have UI effects, and allowing updates to those properties only from UI contexts. The shared property store design appears to us to be a very error-prone design; we would prefer separate property stores or listener registrations for handlers that run on or off the UI thread.

Another problematic idiom, seen only in Subclipse, is code that dynamically checks which thread it is executing on, and optionally redirects a closure to the UI thread if

necessary. Our design does not support these dynamic checks, which are typically found only inside UI libraries themselves. In Subclipse, SWT's `Display.getCurrent()` is used; it returns `null` when executed off the UI thread, and otherwise returns a valid `Display` object. The same method also sets a local boolean indicating whether `Display.getCurrent()` returned `null`, so handling this code is not a simple matter of specializing to a particular `if-then-else` construct.

An idiom responsible for both a number of false reports and for a number of workarounds in our library annotations is to make ad-hoc polymorphic instances of a particular type. By this we mean creating a subtype with a UI-effectful override of a safe method, storing the reference at the (safe) supertype, but only allowing said values to flow to and be used in UI contexts. This frequently occurred with a JDK or Eclipse interface that appeared from documentation to be intended as safe, but some part of JFace or a custom design by a plugin developer co-opted it and treated it as an effect-polymorphic type. Similarly, developers sometimes implement a totally safe subclass of a UI-effectful supertype (often as an anonymous inner class), store references as the supertype, and call methods of said class in safe contexts (only allowing safe subtype instances to flow to those call sites). Checking these uses in general would effectively require allowing every type to take a separate effect parameter for each method. In our evaluation, we typically annotated such classes as effect-polymorphic, annotated each method of those classes as @PolyUIEffect, and added UI instances to the subject programs as necessary.[5] This generally sufficed for UI variants of safe supertypes, unless the problematic class would then inherit from multiple polymorphic types, which our system does not handle. The latter case (safe variants of a UI interface) could be worked around by explicitly introducing a named subtype that declares a safe override, and storing references as the safe subtype (which we did once for HudsonEclipse). We speculate that these designs arise from the designers of the relevant class hierarchies and interfaces not considering the option of using type hierarchies to separate code that must run on the UI thread from code that may run anywhere (including hijacking otherwise safe interfaces). The type hierarchy splitting that fits these cases into our effect system generally seems less error-prone even without our strict effect system checking, because such splitting already introduces some type-based barriers to confusing the calling context of UI code.

A class of idioms that definitely indicates shortcomings of our current effect system is methods requiring qualifier-dependent method effects: effects that depend on the qualifier of one or more effect-polymorphic arguments. There are also some methods whose proper types require a much richer type system. A good example of this is `org.eclipse.swt.custom.BusyIndicator.showWhile()`, whose effect depends on:

- The variant of `Runnable` it receives (@UI or @Safe),
- The calling context (@UIEffect or @SafeEffect), and
- Whether the `Display` it receives is `null`

This method calls the passed `Runnable` on the current thread, displaying a busy cursor on the passed Display if any, and no busy information otherwise. Because it is often used for UI-effectful work, we annotated it @UIEffect, taking a @UI `Runnable`. A better, but still conservative type would be

[5] This is always sound: each instantiation is treated soundly, and @Safe instances may flow into variables for @UI instances.

```
@PolyUIEffect
public static void showWhile(Display display, @PolyUI Runnable runnable);
```

Our type system cannot check calls to this presently because there is no receiver qualifier to tie to the effect at the call site. Checking the method internals would require richer types, including effect refinements based on the `Display`.

4.5 Potential Type System Extensions

Our experiences revealed several ways our type system could be extended to verify more client code.

First, what would be simple polymorphism in the most natural core calculus to write for the polymorphic effect system[6] becomes lightweight dependent effects in the implementation. This happens in cases where a method takes a Runnable of some instantiation and runs it in the current thread. The example we encountered in our evaluation is the `showWhile()` method from the previous section, a good signature for which would be

```
@EffectFromQualOf("runnable")
public static void showWhile(Display display, @PolyUI Runnable runnable);
```

The effect of this runner call will be whatever the effect of the `Runnable`'s `run()` method is. If we could use the same annotation for both qualifiers and method effects, simply specifying the polymorphic qualifier would be sufficient, and the Checker Framework's existing support for polymorphism would handle this. Unfortunately, Java 8's type annotation syntax leaves parsing ambiguities: if @UI applied to both types and methods, there would be no way to disambiguate a use as a method effect annotation from a use as a method return type annotation. Thus we must use different annotations. So to support this type of qualifier-dependent effect, we would need lightweight dependent effects, simply due to limitations of Java's grammar.

A need for dependent effects comes from a pattern seen in Subclipse, where methods either take a boolean argument (`ProgressMonitorDialog.run()` and `Activator.showErrorDialog()`) or call a related method (`Action.canRunAsJob()`) indicating whether or not to fork another thread to execute a Runnable (otherwise the effect of the runner is polymorphic as in the previous example). If the fork flag is true, the provided Runnable must be a @Safe Runnable. So for example, the effect of the main work method for the `Action` class should be (informally):

```
@EffectIfTrueElse(this.canRunAsJob(),@SafeEffect,@UIEffect)
```

indicating that the method must be safe if the method is required to run as a `Job` (a background thread task), and otherwise will run on the UI thread. Checking this then requires executing the `canRunAsJob()` method during type-checking, and ensuring that this computation is independent of subtyping (that the `canRunAsJob()` implementation is `final` when used). Other utility methods take a flag with opposite polarity, but the required type system extension would be the same.

[6] Recall that λ_{UI} is effect-*monomorphic*.

5 Related Work

The most similar work to ours is an open source tool, CheckThread.[7] It is a Java compiler plugin that aims to catch arbitrary concurrency bugs, and includes a `@ThreadConfined("threadName")` annotation similar to our @UIEffect, but supporting arbitrary thread confinement rather than being limited to one distinguished thread. It appears to be an effect system, but the authors never describe it as such in the documentation or code. They allow applying various annotations to whole types as a shorthand for applying an annotation to all methods on a type (similar to our @SafeType and @UIType). However, their system does not support polymorphism, and it is not clear from its documentation if it treats inheritance soundly (from source inspection, it appears not).

Another piece of closely related work is Zhang et al.'s GUI Error Detector [2], which searches for the same errors as our type and effect system via a control flow analysis. Essentially they extract a static call graph of a program from bytecode, and find any call path that begins in non-UI code and reaches a UI method without being "interrupted" by a call to `syncExec()` or `asyncExec()`. The false positive rate from the naïve approach is too high, so they couple this with several heuristics (some unsound, potentially removing correct warnings) to reduce the number of warnings. On four examples from their evaluation, we find all of the bugs they located, plus one more (Section 4.2). They also annotated 19 subject program methods in their evaluation as trusted safe when they performed dynamic thread checks; at least 3 of our false positives would disappear if we suppressed warnings in such methods. Another interesting result of our work is empirical confirmation of the GUI Error Detector's unsound heuristics for filtering reports. GUI Error Detector found most of the bugs our sound approach identified, and the missed bug was due to WALA's defaults for scalable call-graph generation, not due to unsound heuristics. Some idioms, like the global property store design, cause false positives for both techniques.

Our approach has several advantages over Zhang et al.'s approach. Most notably, our technique is sound, while their heuristics may filter out true reports. Our type and effect system is also modular and incremental (we naturally support separate compilation and development) and can reason soundly about code (for example, subclasses) that may not exist yet. The GUI Error Detector on the other hand must have access to all JAR files that would be used by the running application in order to gather a complete call graph, and must rerun its analysis from scratch on new versions. If JARs are unavailable, its unsoundness increases. For performance reasons, their underlying call graph extraction tool (WALA [10]) skips some well-known large libraries — including Swing, meaning that GUI Error Detector misses all callbacks from Swing library code into application code. This results in additional unsoundness, and is the reason GUI Error Detector did not find the drag-and-drop bug in S3Dropbox. Our support for polymorphism is also important: 14 of GUI Error Detector's 24 false positives on Subclipse were because Zhang et al.'s technique does not treat the `Runnable` interface effect-polymorphically.

Our system does have several disadvantages compared to Zhang et al.'s. Our type and effect system requires manual code annotation, requiring many hours for some large projects (though this effort is only required once, with only incremental changes to annotations as the program evolves). Because GUI Error Detector runs on Java bytecode,

[7] `http://checkthread.org/`

it is possible to run the GUI Error Detector on binaries for which source is unavailable, without writing a trusted stub file. It is also possible (though untested) that their approach can handle other JVM languages (such as Scala or Clojure) or multi-language programs. Our type system produces only localized reports; Zhang et al. examine the whole call graph, so a warning's report includes a full potentially-erroneous call sequence. When annotating the sample programs in our evaluation, much of our time was spent manually reconstructing this information.

Sutherland and Scherlis proposed the technique of *thread coloring* [12], which is in some ways a generalization of our UI effect. They permit declaration of arbitrary thread roles and enforcing that methods for certain roles execute only on the thread(s) holding those roles. They use a complex combination of abstract interpretation, type inference, and call-graph reconstruction to reduce annotation burden; the one subject they specify annotation burden for is lower than ours, but they do not provide annotation counts for other subjects. They do not describe their false positive rates. They do annotate AWT and Swing applications successfully; we found AWT and Swing had relatively consistent policies, and expect they would have had more difficulty with Eclipse's SWT and JFace libraries, which were the source of many of our false positives. Like Zhang et al.'s technique, Sutherland's implementation lacks role (effect) polymorphism, which results in an unspecified number of false positives in their evaluation.

There is a long line of work on effect systems, ranging from basic designs [13] to abstract effect systems designed for flexible instantiation [14,15]. Rytz et al. [15] propose an effect-polymorphic function type for reasoning about effects, where for functions of type $T_1 \stackrel{e}{\Rightarrow} T_2$ the effect of invoking it is the join of the concrete effect e with the effect of the argument T_1, implying that the function may call T_1 (if the argument is itself a function). This is similar to what we would need to support qualifier-dependent effects (e.g., `showWhile` in Section 4.5). Other effect systems are designed to reason about more general safe concurrency properties [16,17], but we are the first to build a polymorphic effect system for the issue of safe UI updates. Another approach would be to encapsulate UI actions within a monad, since every effect system gives rise to a monad [18]. Functional languages such as Haskell use monads to encapsulate many effects, and in fact some Haskell UI libraries use a UI monad to package UI updates safely (e.g., Phooey [19]). Another use of monads in functional languages is to support software transactional memory [20,21], including a strong separation between data accessed inside or outside a transaction. Viewing the closures run on the UI thread as *UI transactions*, our type system enforces a weakly atomic [22] form of transactions, where UI elements are guaranteed to be transaction-only but non-UI elements have no atomicity guarantees.

6 Conclusion

In almost every UI framework, it is an error for a background thread to directly access UI elements. This error is pervasive and severe in practice. We have developed an approach — a type and effect system — for preventing these errors that is both theoretically sound and practical for real-world use. We have proven soundness for a core calculus λ_{UI}. Our implementation, $\mathsf{Java}_{\mathsf{UI}}$, is both precise and effective: in 8 projects

totalling over 140,000 LOC, Java$_{UI}$ found all known bugs with only 30 spurious warnings, for a modest effort of 7.4 annotations per 1000 LOC on average.

We have identified error-prone coding idioms that are common in practice and explained how to avoid them. We also identified application patterns that Java$_{UI}$ cannot type check that will probably be issues for other effect systems applied to existing code: ad-hoc effect polymorphism, value-dependent effects, and data structures mixing callbacks with different effects. These idioms suggest improvements to existing code (such as segregating callbacks with different known effects) and profitable extensions to effect systems.

Acknowledgements. We thank the anonymous referees for their helpful comments, and Sai Zhang for helpful discussions and running GUI Error Detector on additional subject programs for us. This work was funded in part by NSF grant CNS-0855252 and DARPA contracts FA8750-12-C-0174 and FA8750-12-2-0107.

References

1. Ingalls, D.H.H.: The Smalltalk-76 Programming System: Design and Implementation. In: POPL (1978)
2. Zhang, S., Lü, H., Ernst, M.D.: Finding Errors in Multithreaded GUI Applications. In: ISSTA (2012)
3. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Inf. Comput. 115(1), 38–94 (1994)
4. Gordon, C.S., Dietl, W.M., Ernst, M.D., Grossman, D.: Java$_{UI}$: Effects for Controlling UI Object Access. Technical Report UW-CSE-13-04-01, University of Washington (2013)
5. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical Pluggable Types for Java. In: ISSTA (2008)
6. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K., Schiller, T.: Building and Using Pluggable Type-Checkers. In: ICSE (2011)
7. Foster, J.S., Fähndrich, M., Aiken, A.: A Theory of Type Qualifiers. In: PLDI (1999)
8. Emir, B., Kennedy, A., Russo, C., Yu, D.: Variance and Generalized Constraints for $C^\sharp$ Generics. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 279–303. Springer, Heidelberg (2006)
9. Wheeler, D.A.: SLOCCount, `http://www.dwheeler.com/sloccount/`
10. WALA, `http://wala.sourceforge.net`
11. Subclipse Issue 889, `http://subclipse.tigris.org/issues/show_bug.cgi?id=889`
12. Sutherland, D.F., Scherlis, W.L.: Composable Thread Coloring. In: PPoPP (2010)
13. Lucassen, J.M., Gifford, D.K.: Polymorphic Effect Systems. In: POPL (1988)
14. Marino, D., Millstein, T.: A Generic Type-and-Effect System. In: TLDI (2009)
15. Rytz, L., Odersky, M., Haller, P.: Lightweight Polymorphic Effects. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 258–282. Springer, Heidelberg (2012)
16. 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)
17. Kawaguchi, M., Rondon, P., Bakst, A., Jhala, R.: Deterministic Parallelism via Liquid Effects. In: PLDI (2012)
18. Wadler, P.: The Marriage of Effects and Monads. In: ICFP (1998)
19. Phooey UI Framework, `http://www.haskell.org/haskellwiki/Phooey`

20. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC (2003)
21. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP (2005)
22. Blundell, C., Lewis, E., Martin, M.: Subtleties of Transactional Memory Atomicity Semantics. Computer Architecture Letters 5(2) (2006)

The Billion-Dollar Fix

Safe Modular Circular Initialisation with Placeholders and Placeholder Types

Marco Servetto, Julian Mackay, Alex Potanin, and James Noble

Victoria University of Wellington
School of Engineering and Computer Science
{servetto,mackayjuli,alex,kjx}@ecs.vuw.ac.nz

Abstract. Programmers often need to initialise circular structures of objects. Initialisation should be safe (so that programs can never suffer null pointer exceptions or otherwise observe uninitialised values) and modular (so that each part of the circular structure can be written and compiled separately). Unfortunately, existing languages do not support modular circular initialisation: programmers in practical languages resort to Tony Hoare's "Billion Dollar Mistake": initialising variables with nulls, and then hoping to fix them up afterward. While recent research languages have offered some solutions, none fully support safe modular circular initialisation.

We present *placeholders*, a straightforward extension to object-oriented languages that describes circular structures simply, directly, and modularly. In typed languages, placeholders can be described by *placeholder types* that ensure placeholders are used safely. We define an operational semantics for placeholders, a type system for placeholder types, and prove soundness. Incorporating placeholders into object-oriented languages should make programs simultaneously simpler to write, and easier to write correctly.

1 Introduction

Imagine writing the top level of a simple web application, with a database access `DBA` component, a `Security` component, an `SMS` component able to send SMS text messages and finally a `GUI` component for user interaction. The security component needs to access the database (to retrieve user authorisation records), while the database needs to access the security component (to ensure users only access data they are authorised to view); the `GUI` component needs access to the `SMS` component (to send messages) and the `SMS` component needs access to the database to log received/sent messages; finally the database needs to update the `GUI` component whenever a change happens (for example to display received SMS messages). This system has a number of circular dependencies: the database needs the security system, which needs the database, which needs the security system; the `GUI` component needs the `SMS` component, which needs the database access, which needs (to update) the `GUI` component; and on *ad infinitum*.

How can programs initialise such a structure? The obvious code is obviously wrong:

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 205–229, 2013.

first attempt
```
Security s=new Security(dba);
DBA dba=new DBA(s,gui);
SMS sms=new SMS(s,dba);
GUI gui=new GUI(sms,dba,s);
```

attempting to initialise the security system with the database before the database itself is constructed. In languages designed in the last twenty years or so, this "uninitialised variable" error should be caught statically or dynamically; in older languages, this will be caught as a null pointer exception (if we are lucky) or by initialising the security system with the contents of the uninitialised `dba` variable (if we are not).

The traditional solution relies critically on Tony Hoare's "Billion Dollar Mistake": null pointers [12]. We initialise the security system with a null pointer, instead of a database, then initialise the database with the now-extant security subsystem, and then use a setter method to link the database back to the security system:

nulls and setters
```
Security s=new Security(null);
DBA dba=new DBA(s,null);
s.setDatabase(dba);
SMS sms=new SMS(s,dba);
GUI gui=new GUI(sms,dba,s);
sms.setGUI(gui);
dba.setGUI(gui);
```

More sophisticated versions of this approach may use dependency injection frameworks, writing XML configuration files rather than code, decoupling configuration from the code itself; or (monadic) option types rather than raw nulls, avoiding null pointer errors at the cost of enforcing tests of every access to every potentially uninitialised variable throughout the program.

1.1 Placeholders

We address this problem by introducing *placeholders*. A placeholder, as the name suggests, is a proxy or a stand-in for an uninitialised, as yet nonexistent object. Placeholders are created in *placeholder declarations*. A single placeholder declaration can declare and initialise the entire GUI/Database/SMS system:

placeholders
```
Security s=new Security(dba),
DBA dba=new DBA(s,gui),
SMS sms=new SMS(s,dba),
GUI gui=new GUI(sms,dba,s);
```

Placeholder declarations differ from the Java-style declarations shown earlier in two ways. Syntactically, a series of initialisation clauses are separated with commas (`,`). Programs can pass variables — placeholders, in fact — declared anywhere in the same placeholder assignment statement as arguments to any constructors within the same statement. In the code above, these placeholders are shown with a red background.

Semantically, placeholder declarations are evaluated in three phases (see Fig.1). First, a placeholder is created and bound to each declared placeholder variable. Second, the right-hand sides of each declaration (the initialisers) are run in turn, top to bottom, creating the actual objects that will be used in the rest of the program. Third, all pointers

to placeholders are redirected to point to the objects whose places they are holding. This redirection includes all variable bindings, so at the end of the third phase, all the variables are bound to the actual objects created in the second stage. At this point, execution of the placeholder declaration is complete, and the program can continue.

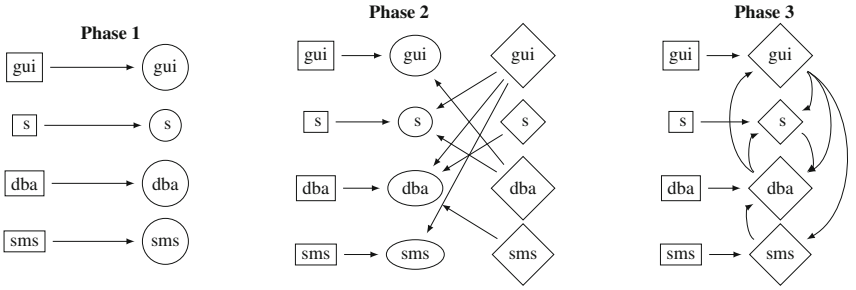


Fig. 1. Initialisation with placeholders

A key feature of placeholders and placeholder declarations is that they have minimal impact on the rest of the program. Placeholders are lexically scoped, limited in extent to the execution of the enclosing placeholder declaration. Placeholder declarations as a whole have well defined imperative semantics — important if any of the initialisers have side effects — as they are simply executed in the order in which they are written.

Placeholders are also flexible. For example, to avoid hard wiring classes into client code, programmers and languages are adopting the object-algebra style, where a *platform* object provides a single point of configuration [15,3]. Client code requests classes from the platform object, and then instantiates those objects indirectly:

object algebra

```
Platform p = ...; // p is a factory aka object algebra

Security s=p.makeSecurity(dba),
DBA dba=p.makeDBA(s,gui),
SMS sms=p.makeSMS(s,dba),
GUI gui=p.makeGUI(sms,dba,s);
```

This more complex creation style also works well with placeholders — the only difference is that placeholders are passed into factory methods [7], rather than directly into constructors. Overall, placeholders support circular initialisation of independent, encapsulated, modules, without any null pointers — the components to be created and initialised need only to present a factory interface where placeholders may be passed into some arguments. More than this: because placeholders can displace nulls for field initialisation, a programming language with placeholders can do without traditional Hoare-style null values entirely.

Unfortunately, placeholders are not by themselves a complete solution to the circular initialisation problem. Placeholders can replace nulls, allowing programs to create circular structures idiomatically. Placeholders are not accessible *outside* the lexical scope of their placeholder declaration, but placeholders have to be accessible *inside* the scope of their declarations, so that they can be used to configure other components. The problem

is that placeholders are placeholders, not actual objects, indeed the object to which they will refer may not have been created when the placeholder is used. Here's a slightly modified example, where the security system is configured last, and where it attempts to retrieve the DBA component via the GUI component, rather than from the dba variable.

placeholders

```
DBA dba=new DBA(s,gui),
SMS sms=new SMS(s,dba),
GUI gui=new GUI(sms,dba,s),
Security s=new Security(gui.getDBA());
```

To a first approximation, this code could reasonably be expected to work: when the GUI component is created, in phase 2 of the execution of the placeholder declaration, it will have received the placeholder for the DBA component, so a getDBA accessor method called on the DBA *object* should just return that placeholder. Indeed, a call on the DBA object would do just that. Unfortunately, at phase 2, all references are to *placeholders*, not to objects — note that the first three lines happily initialise the DBA, SMS, and GUI components with s, which must be a placeholder for the security component as the security component has not yet been created.

So what can we do when a method is requested on a placeholder — effectively a nonexistent object — rather than the object that has yet to come into being? Well, in the absence of time-travel, we treat this as a programmer error, and throw a *PlaceholderException*. Placeholders, after all, are not objects, they are just placeholders for objects. This rationale is also why we do not replace references to placeholders until all the object are created and initialised: we wish to avoid partially initialised objects as much as possible.

This means we have jumped out of the frying pan and into the fire. Rather than having just one distinguished "null" pseudo-object that can be used to manage object initialisation (and for many other purposes) we have created a vast number of placeholder pseudo-objects, access to any one of which terminates program execution with a PlaceholderException just as surely as a null terminates a program with a NullPointerException.

1.2 Placeholder Types

To solve the problem of the PlaceholderExceptions (that replace NullPointerExceptions), we provide a static type system with support for *Placeholder Types*. This type system guarantees that references to (potential) placeholders may only be accessed once they have been replaced with actual objects.

Placeholder types are a small addition to standard Java-like type systems. A reference of placeholder type may refer either to objects or to placeholders of the underlying object type — we will write C′ for the placeholder type corresponding to an object type C. The types in the remainder of the system (we term them "object types" where it is necessary to distinguish) are essentially standard Java types: they accept objects, but do not accept placeholders; nor do they accept **null** pointers — indeed, there are no **null** values at all in our formal system. A reference of object type always refers to an instance of the declared type, never to a placeholder.

Because of the limited scope and lifetimes of the placeholders themselves, placeholder types are required only in a very small part of most programs — the placeholder declara-

tions, constructors, and factory methods used to create and initialise components. Within the actual body of a placeholder declaration, the declared variables are all interpreted as having placeholder types, while after the end of the placeholder declaration — but while those variables are still in scope — the declared variables have object types. This follows directly from the semantics of the placeholder declarations: in phase 1, placeholders are bound to all the declared variables, while at the end of phase 3, all those placeholders have been replaced by actual objects, so that variables that were holding placeholders now refer to those actual objects.

placeholder types

```
DBA dba=new DBA(s,gui),
SMS sms=new SMS(s,dba),
GUI gui=new GUI(sms,dba,s),
  // ''gui'' has placeholder type ''GUI''  within
  // placeholder declaration statement
Security s=new Security(gui.getDBA());
  // gui.getDBA() here is a type error,
  // cannot request methods on placeholders
gui.getDBA();
  // after the end of the placeholder declaration
  // gui has object type ''GUI'' --- it cannot be
  // a placeholder, so method requests are permitted
```

To prevent PlaceholderExceptions from being raised when methods are requested from placeholders (instead of objects) we forbid placeholder types in the receiver position ("before the dot") of any method requests. This rule prevents calls like "`gui.getDBA())`" within the body of the placeholder declaration itself (before the first semicolon "`;`"), when "`gui`" has placeholder type "`GUI'`". After the end of the placeholder declaration — but within the block where the declared names are visible — "`gui`" has object type "`GUI`" and may receive method requests.

To keep the overall system simple, we further restrict placeholder types. Placeholder types may only appear as parameters or return values from methods, or parameters to constructors: they may not appear as types of objects' fields. The aim here is to ensure that placeholders do not escape from the lexical context of their placeholder declarations, so that all the objects created with those placeholders are fully initialised at the end of their placeholder declaration. These restrictions are liberal enough to permit many kinds of factory methods and object algebras. For example, here is code implementing the `makeSecurity` method in the `Platform` object factory (object algebra) discussed earlier:

object algebra

```
class Platform {//...
  Security makeSecurity(DBA' dba) {
    return new Security(dba);
  }
}
```

Here the `makeSecurity` factory method takes a `DBA'` placeholder type and passes it as an argument to the underlying constructor. This is permitted by the rules on the use of placeholder types, while invoking a method on `dba`, or storing it into a field is not permitted, as that could lead to a PlaceholderException, sooner or later.

These rules are necessarily subtle, especially when objects are created that are initialised by placeholders. These partially initialised objects will only become fully initialised when those placeholders themselves become fully initialised. Partially initialised objects do not have placeholder types — in particular, they may be returned as object types — but they are treated as if they were placeholders until they are returned. Consider the following version of the Platform class:

partial initialisation

```
class DifferentPlatform {
  Security makeSecurity(DBA' dba) {
    ExternalSecurity x = new ExternalSecurity(dba);
    x.validate();
    //x.validate here is a type error
    //cannot request methods on partially initialised objects
    if (/*high security needed*/)return highSecurity(x);
    else return x;
    //this is not a type error
    //may return partially initialised objects
  }
}
```

Here we create a new object, initialised with a placeholder, and store that in the local variable `x`. This variable is not a placeholder type — we can return it as an object type — even though it refers to a partially initialised object. We know this object is partially initialised because its value comes from an expression that has a placeholder (`dba`) passed in as an actual argument. Within this method, we treat `x` as a placeholder, so we can pass this partially initialised object as a placeholder typed argument to another method (as `Security highSecurity(Security' s)`), but we cannot call methods upon it.

Partially initialised objects are safe because they are treated as if they were placeholders until they are fully initialised. Partially initialised objects can only be passed as placeholder typed arguments, and so they will be treated as full placeholders by any methods into which they are passed. When partially initialised objects are returned from methods (as object types) if that method was called with placeholder arguments, then that returned value will also be treated as partially initialised.

Finally, in order to be able to actually initialise fields of objects, we have to treat placeholder types in constructors slightly differently from elsewhere (of course, constructors already have special privileges in Java-like languages, notably to be able to initialise final fields. The special case within constructors is that we can initialise a field of object type `T` with an expression of placeholder type `T'` — because of our restrictions on placeholder types, any such placeholder must have been passed in as an argument to that constructor.

class decl

```
class Security {
  DBA dbaComponent;
  Security(DBA' dba) {this.dbaComponent=dba;}
}
```

To maintain soundness, we cannot read values back from fields in constructors, otherwise we could attempt to request a method from a placeholder stored in a field with

p	$::= \overline{cd}\ \overline{id}$	program
cd	$::= \texttt{class}\ C\ \texttt{implements}\ \overline{C}\ \texttt{\{}\ \overline{fd}\ k\ \overline{md}\ \texttt{\}}$	class declaration
id	$::= \texttt{interface}\ C\ \texttt{extends}\ \overline{C}\ \texttt{\{}\ \overline{mh;}\ \texttt{\}}$	interface declaration
fd	$::= C\ f\texttt{;}$	field declaration
k	$::= C\texttt{(}\ C'_1\ f_1\texttt{,}\ \ldots\texttt{,}\ C'_n\ f_n\texttt{)\{ this.}f_1\texttt{=}f_1\texttt{; }\ldots\texttt{; this.}f_n\texttt{=}f_n\texttt{;\}}$	constructor
mh	$::= T\ m\texttt{(}\overline{T\ x}\texttt{)}$	method header
md	$::= mh\ e$	method declaration
e	$::= x \mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid \texttt{new}\ C\texttt{(}\overline{e}\texttt{)} \mid e\texttt{.}f \mid e_1\texttt{.}f\texttt{=}e_2 \mid \iota$	expressions
	$\mid xe_1\texttt{,}\ \ldots\texttt{,}\ xe_n\texttt{;}\ e$	place holder declaration
xe	$::= T\ x = e$	variable initialisation
v	$::= \iota \mid x$	value
μ	$::= \overline{\iota \mapsto C(f_1 = v_1, \ldots, f_n = v_n)}$	memory
T	$::= C \mid C'$	type

Fig. 2. Syntax

an object type which ostensibly does not permit placeholders. On the other hand, since placeholder types can refer to objects (of the appropriate type) as well as placeholders, this constructor and the above object factory can always be called passing in actual objects, rather than placeholders, if the programmer has them to hand.

1.3 Contributions

This paper makes two main contributions: first, placeholders and placeholder declarations, and second, placeholder types. Placeholders support idiomatic, modular, circular initialisations of complex object structures, while placeholder types ensure placeholders are only used safely, and so will never cause a PlaceholderException.

The rest of this paper is structured as follows. Section 2 presents the FJ′ language and its formalisation and Section 3 gives the details of the type rules and states the main soundness theorems. Section 4 demonstrates the expressiveness of FJ′ by presenting a selection of more complex examples, and then Section 5 discusses implementation considerations for supporting placeholders in a Java-like setting. Section 6 overviews related work and Section 7 concludes. The accompanying technical report presents the proofs for our formal system and a more extensive discussion of placeholders [17].

2 Syntax and Semantics of FJ′

Syntax

The syntax of FJ′ is shown in Figure 2. We assume countably infinite sets of variables x, object identifiers ι, class or interface names C, method names m, and field names f. As in FJ [13] variables include the special variable **this**.

Classes and interfaces. A program p is a set of class and interface declarations. A class declaration consists of a class name followed by the set of implemented interfaces, the sequence of field declarations, a conventional constructor and the set of method declarations. To keep the presentation focused on the problem of circular initialisation, we do not consider class composition operators like the *extends* operator in Java.

An interface declaration consists of an interface name followed by the sets of extended interfaces and method headers. As one can see, for simplicity, we use interfaces to provide subtyping, i.e. there is no subtype relation between classes. Field declarations are as in FJ. To simplify the formalisation, constructors (k) are standard conventional FJ constructors, taking exactly one parameter for each field and only initialising fields. Method declarations are composed of a method header and a body. The method header is as in FJ. Since only fully initialised objects can be receivers, the implicit parameter **`this`** is always of a fully initialised object type. As in FJ, method bodies are simply expressions. We omit the return keyword in the formal definition but we insert it in the appropriate places in the examples to aid readability.

Expressions. Expressions are variables, method or constructor calls, field access, field update, object identifiers (within run time expressions), and finally, placeholder declarations. Variables can be declared in method headers or inside expressions. An expression is well-formed only if the same variable is not declared twice. Hence, we have no concept of variable hiding. Placeholder declarations are a sequence of variable initialisation clauses separated by comma (`,`) and an expression terminated by semicolon (`;`). Note that the order of variable initialisations is relevant since it induces the order of execution for the sub-expressions. The order is also relevant in sequences of field declarations and parameter declarations.

Syntax for placeholders. In a closed expression an occurrence of x is a placeholder only if it is inside a placeholder declaration declaring local variable x, and before the semicolon. That is, any placeholder declaration with a variable initialisation clause $T\ x$=... can contain the placeholder x inside its initialisation expressions, and (as usual) the variable x inside its terminating expression.

Values and memory. Values are object identifiers ι or placeholders x. Note that values do not include **`null`** and we have no default initialisation. Placeholders are replaced by object identifiers when placeholder declarations are reduced. Thus final values are only object identifiers.
A memory is a finite map from object identifiers to records annotated with a class name. For example, a computation over the program "**`class`** `C`**`{`**`C f;C(C' f)`**`{this.`**`f=f;}}`", can produce the memory "$\iota_1 \mapsto$ `C(f=y)`, $\iota_2 \mapsto$ `C(f=`ι_1`)`, $\iota_3 \mapsto$ `C(f=`ι_3`)`", containing three object identifiers for objects of class `C`: }

- ι_1 refers to an object containing a single field named `f` pointing to the placeholder `y`,
- ι_2 refers to an object containing a single field named `f` pointing to ι_1. Thus, ι_1 and ι_2 denote two *partially initialised objects*: objects containing a placeholder or another partially initialised object where a *fully initialised object* is expected.
- ι_3 denotes an object containing a single field named `f`, referencing to ι_3 itself. Thus ι_3 is a fully initialised object.

A memory is well-formed with respect to a program if the fields of the records are exactly the fields declared inside the corresponding classes. (Note that a well formed memory may not be well typed, see rule (MEM-OK)).

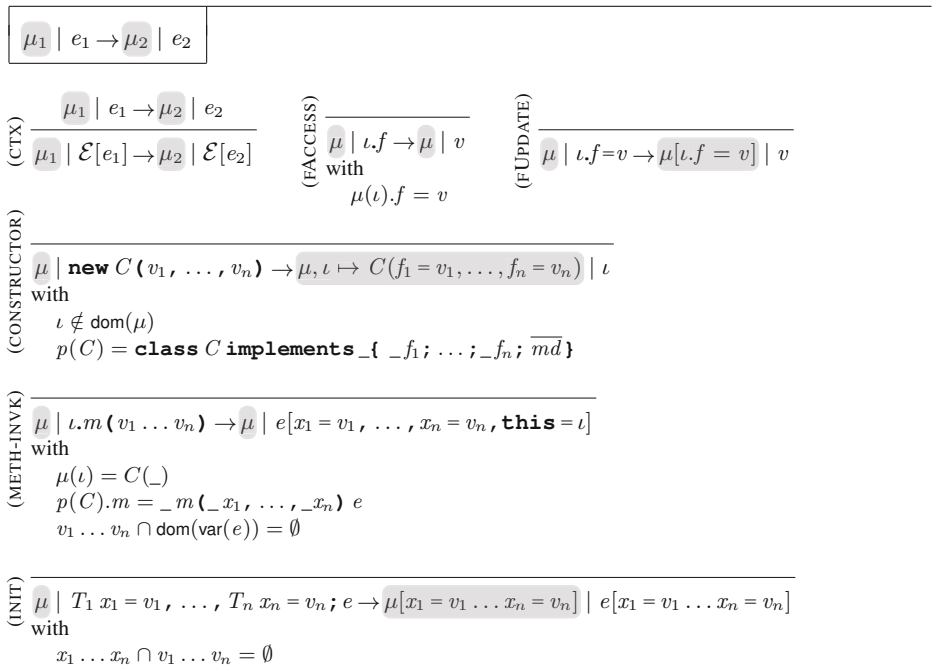


Fig. 3. Reduction rules using memory

Types. are composed of a class name C and an optional quote ($'$) sign. We assign type C' (*placeholder type*) to placeholders and type C (*object type*) to objects of class C. C is a subtype of C'.

Expressions of type C reduce to objects (either partially or fully initialised), while expressions of type C' can also reduce to placeholders. Fields can only be of object type C, and field access is always guaranteed to return a fully initialised object.

Subtyping

Our subtyping is the normal Java subtyping, with the addition that a placeholder type is a subtype of the corresponding object type. Formally:

- $C_1 \leq C_2$ if $p(C_1) = \mathtt{class}\ C_1\ \mathtt{implements}\ C, \mathtt{\_\{\ \_\}}$ and $C \leq C_2$
- $C_1 \leq C_2$ if $p(C_1) = \mathtt{interface}\ C_1\ \mathtt{extends}\ C, \mathtt{\_\{\ \_\}}$ and $C \leq C_2$
- $T \leq T$
- $C_1' \leq C_2'$ and $C_1 \leq C_2'$ if $C_1 \leq C_2$

Reduction

Reduction is defined in the conventional way as an arrow over pairs consisting of a memory and an expression. The only novelty is rule (INIT). To improve readability we will mark memory in grey. A pair $\mu \mid e$ is well-formed only if all the placeholders contained in the memory μ are bound by the local variables declared inside the expression e; that is, a memory with placeholders without an associated expression is meaningless.

We omit the formal definition of the evaluation context, assuming a standard deterministic left-to-right call-by-value reduction strategy.

Figure 3 defines the reduction arrow. Rule (CTX) is standard, however note that alpha-conversion can be needed to ensure the well-formedness of the resulting expression. Rule (FACCESS) models conventional field access. It extracts the value of field f from object ι using the notation $\mu(\iota).f$. Rule (CONSTRUCTOR) is the standard reduction for constructor invocations, and rule (METH-INVK) models a conventional method call. We assume a fixed program p and we use notation $p(C).m$ to extract the method declaration.

We use the notation $e[x_1 = v_1, \ldots, x_n = v_n]$ for variable substitution, that is, we simultaneously replace all the occurrences of x_i in e with v_i. The last side condition ensures that no placeholder inside the set of values $v_1 \ldots v_n$ is accidentally captured when injected inside the expression e. Alpha-conversion can be used to satisfy this side condition. From any expression it is possible to extract a map from placeholders to their declared type, denoted by $\mathsf{var}(e)$. Formally:

$\mathsf{var}(\iota) = \emptyset, \quad \mathsf{var}(x) = \emptyset, \quad \mathsf{var}(e.m(\overline{e})) = \mathsf{var}(e), \mathsf{var}(\overline{e}),$ and

$\mathsf{var}(T_1\ x_1 = e_1, \ldots, T_n\ x_n = e_n; e_0) = x_1 : T_1, \ldots, x_n : T_n, \mathsf{var}(e_0), \ldots, \mathsf{var}(e_n).$

Rule (INIT) reduces placeholder declarations. Just as $e[\overline{x = v}]$ denotes simultaneous replacement of variables with values, we use the analogous notation $\mu[\overline{x = v}]$ to denote simultaneous replacement of placeholders with values. Formally: $[\overline{x = v}] = \emptyset$ and $(\mu, \iota \mapsto C(f_1 = v_1, \ldots, f_n = v_n))[\overline{x = v}] = \mu[\overline{x = v}], \iota \mapsto C(f_1 = v_1[\overline{x = v}], \ldots, f_n = v_n[\overline{x = v}])$.

Note how a normal variable declaration is just a special case of our placeholder declaration, where only one local variable is declared, the placeholder is not used and, thus, the placeholder replacement is an empty operation. Rule (INIT)'s side condition verifies that meaningless terms like `T x=x;x` are stuck.

To understand the details of the semantics of placeholders, consider the following example:

FJ−
```
class A{B myB;A(B' myB){this.myB=myB;}}
class B{A myA;B(A' myA){this.myA=myA;}}
...
  A a=new A(b), B b=new B(a); a
```

We show how to evaluate that expression in the empty memory.

[0]	$\emptyset$	`A a=new A(b), B b=new B(a); a`	Starting point
[1]	$\iota_1 \mapsto$ `A(myB=b)`	`A a=`ι_1`, B b=new B(a); a`	(CTX) + (CONSTRUCTOR)
[2]	$\iota_1 \mapsto$ `A(myB=b)` $\iota_2 \mapsto$ `B(myA=a)`	`A a=`ι_1`,B b=`ι_2`; a`	(CTX) + (CONSTRUCTOR)
[3]	$\iota_1 \mapsto$ `A(myB=`ι_2`)` $\iota_2 \mapsto$ `B(myA=`ι_1`)`	ι_1	(INIT)

- We start in state [0], and in the first step an instance of class `A` is created.
- In state [1], the memory contains a partially initialised object of type `A` containing placeholder `b` instead of a reference to an object of type `B`. The reference corresponding to the placeholder `b` is still unknown. Placeholders are not objects, and thus there is no reference in the memory pointing directly to a placeholder. You can now see that values are object identifiers ι or placeholders x.
- In the second step the instance of class `B` is created. In state [2] ι_1 and ι_2 are partially initialised objects. Note how placeholders inside the memory are bound by the local variables declared inside the placeholder declaration.

– The third step concludes the initialisation, and in state [3] ι_1 and ι_2 are fully initialised objects. The placeholders are replaced by objects when placeholder declarations are reduced. That is, when the control reaches the semicolon, all the placeholders declared in that placeholder declaration are *consumed*; every occurrence of such a placeholder in the memory is replaced with the corresponding value.

3 Type System of FJ′

Our definition of reduction introduces two stuck situations that are novel in FJ′: (1) method call (or field access) over a placeholder receiver, and (2) declarations of the form `T x = x`.

Our placeholder type system must prevent both these situations. For the first case, while it is clear that we need to forbid method invocation or field access on placeholders, there could be different strategies to ensure safety while manipulating partially initialised instances (objects containing placeholders in their reachable object graph). We believe that in this case the simplest solution is also the right one: **to forbid any method invocation or field access over partially initialised objects**, as we do for placeholders.

The solution for the second case relies on the distinction of placeholder types and object types: it is not possible to initialise a local variable or method parameter of object type with an expression of placeholder type, while a constructor **can** initialise a field with a placeholder.

Limitations over Parameters and Conventional Expression Typing Rules
We permit method parameters to have placeholder type — any factory method allowing circular initialisation needs to have at least one parameter with placeholder type. To ensure that a placeholder is never dereferenced, a method invocation must provide a fully initialised object for any parameter of object type (including the receiver), while either fully initialised objects or placeholders may be provided to parameters of placeholder type.

In FJ′, any closed expression of type C is guaranteed to reduce to a fully initialised object, while any expression using a placeholder could reduce to a partially initialised object. We say that an expression using a placeholder *depends* on that placeholder. Local variable declarations makes the reasoning a little bit more involved: any expression using a variable that depends on a placeholder, also depends any other placeholders that their placeholder declaration depends on. Consider the following code:

```
class C{ C x; C(C' x){this.x=x;} void m(){...} }
void example(C' x){
  //new C(x).m();//wrong: receiver depends from placeholder x

  C y=new C(x);
  //y.m();//wrong: receiver depends from placeholder x through variable y

  C z=new C(z);
  z.m();//correct

  C z1=new C(z2), C z2=new C(x);
  //z1.m();//wrong: z1 is declared together with z2, both depend on x
  }
```

- FJ′ -

$\Gamma; \Sigma \vdash e : T$

(VAR-T) $\dfrac{}{\Gamma; \Sigma \vdash x : \Gamma(x)}$

(ADDR-T) $\dfrac{}{\Gamma; \Sigma \vdash \iota : \Sigma(\iota)}$

(METH-INVK-T) $\dfrac{\begin{array}{c}\mathsf{objectTypes}(\Gamma); \Sigma \vdash e_0 : C_0 \\ \forall i \in 1..n \text{ such that } T_i = C_i : \ \mathsf{objectTypes}(\Gamma); \Sigma \vdash e_i : \_ \leq C_i \\ \forall i \in 1..n \text{ such that } T_i = C_i' \ : \ \Gamma; \Sigma \vdash e_i : \_ \leq C_i'\end{array}}{\Gamma; \Sigma \vdash e_0.m(e_1, \ldots, e_n) : T}$
with
$p(C_0).m = T\ m(T_1\ x_1, \ldots, T_n\ x_n)\ \_$

(NEW-T) $\dfrac{\forall i = 1..n : \quad \Gamma; \Sigma \vdash e_i : \_ \leq C_i'}{\Gamma; \Sigma \vdash \mathbf{new}\ C(e_1 \ldots e_n) : C}$
with
$p(C) = \mathbf{class}\ C\ \mathbf{implements}\ \_\{\ C_1\ \_; \ldots; C_n\ \_; \overline{md}\}$

(FACCESS-T) $\dfrac{\mathsf{objectTypes}(\Gamma); \Sigma \vdash e_0 : C_0}{\Gamma; \Sigma \vdash e_0.f : C_1}$
with
$p(C_0) = \mathbf{class}\ C_0\ \mathbf{implements}\ \_\{\ \_\ C_1\ f; \_\}$

(FUPDATE-T) $\dfrac{\begin{array}{c}\Gamma; \Sigma \vdash e_0 : C_0 \\ \mathsf{objectTypes}(\Gamma); \Sigma \vdash e_1 : C_1\end{array}}{\Gamma; \Sigma \vdash e_0.f = e_1 : C_1}$
with
$p(C_0) = \mathbf{class}\ C_0\ \mathbf{implements}\ \_\{\ \_\ C_1\ f; \_\}$

Fig. 4. Typing rules for expressions

FJ′ constructors can be called using expressions of placeholder type as parameters, and then initialise fields of object type. Consider the following code:

-FJ′-
```
class A { B myB; A(B′ myB){this.myB=myB;} }
class B { A myA; B(A′ myA){this.myA=myA;} }
...
  A a = new A(b), B b = new B(a);
```

Observe that we invoked the constructors of A and B by passing placeholders that then initialise fields of object type.

Typically, a constructor for A produces a fully initialised instance of A when a fully initialised instance of B is provided. FJ′ extends this behaviour so that a *partially* initialised object is returned instead of a *fully* initialised object when either a partially initialised or a placeholder argument is supplied to a constructor call.

Figure 4 shows the rules for expressions that capture the discussed discipline. The typing judgement for expressions uses the following environments:
$\Gamma ::= \overline{x{:}T}$ variable environment
$\Sigma ::= \overline{\iota{:}C}$ memory environment
Variable environment Γ is a map from variable names to types. Memory environment Σ is a map that for any location stores its class name C. A type judgement is of the form $\Gamma; \Sigma \vdash e : T$, where Γ is needed to type expressions with free variables or placeholders, and Σ is needed to type expressions with locations.

Rules (VAR-T) and (ADDR-T) are standard and straightforward. Rule (METH-INVK-T) uses notation $\mathsf{objectTypes}(\Gamma)$ to restrict the environment to the object types. Formally: $\mathsf{objectTypes}(x{:}C) = x{:}C$ and $\mathsf{objectTypes}(x{:}C') = \emptyset$. Every expression well typed in

$$\boxed{\Gamma; \Sigma \vdash e : T}$$

$$\text{(V-DEC-N)}\ \frac{\begin{array}{c}\forall i = 1..n \text{ with } T_i = C_i : \quad \mathsf{objectTypes}(\Gamma); \Sigma \vdash e_i : \_ \leq T_i \\ \forall i = 1..n \text{ with } T_i = C_i' \ : \quad \Gamma; \Sigma \vdash e_i : \_ \leq T_i \\ \Gamma, x_1{:}T_1, \ldots x_n{:}T_n; \Sigma \vdash e : T\end{array}}{\Gamma; \Sigma \vdash T_1\ x_1 = e_1\texttt{,}\ \ldots\texttt{,}\ T_n\ x_n = e_n\texttt{;}\ e : T}$$

$$\text{(V-DEC-C)}\ \frac{\begin{array}{c}\forall i = 1..n : \quad \mathsf{objectTypes}(\Gamma), x_1{:}C_1' \ , \ldots x_n{:}C_n' \ ; \Sigma \vdash e_i : \_ \leq C_i \\ \Gamma, x_1{:}C_1, \ldots x_n{:}C_n; \Sigma \vdash e : T\end{array}}{\Gamma; \Sigma \vdash C_1\ x_1 = e_1\texttt{,}\ \ldots\texttt{,}\ C_n\ x_n = e_n\texttt{;}\ e : T}$$

$$\text{(V-DEC-O)}\ \frac{\begin{array}{c}\forall i = 1..n : \quad \Gamma, x_1{:}C_1' \ , \ldots x_n{:}C_n' \ ; \Sigma \vdash e_i : \_ \leq C_i \\ \Gamma, x_1{:}C_1, \ldots x_n{:}C_n; \Sigma \vdash e : T \\ \Gamma, x_1{:}C_1' \ , \ldots x_n{:}C_n' \ ; \Sigma \vdash e : \_\end{array}}{\Gamma; \Sigma \vdash C_1\ x_1 = e_1\texttt{,}\ \ldots\texttt{,}\ C_n\ x_n = e_n\texttt{;}\ e : T}$$

Fig. 5. Typing rules for placeholder declarations

$\mathsf{objectTypes}(\Gamma)$ denotes a fully initialised object. Thus, rule (METH-INVK-T) requires (premise one) the receiver and (premise two) all actual arguments to formal parameters of object type to be fully initialised objects, and (premise three) permits actual arguments to formal parameters of placeholder type to be placeholders or partially initialised objects and well as fully initialised objects. Note how we use notation $\Gamma; \Sigma \vdash e : T_1 \leq T_2$ as a shortcut for $\Gamma; \Sigma \vdash e : T_1$ and $T_1 \leq T_2$. Note that the last (_) in the side condition of (METH-INVK-T) could be either a method body or a semicolon, depending on C_0 being a class or an interface. Rule (NEW-T) is conventional but accepts arguments of placeholder types. Rules (FACCESS-T) and (FUPDATE-T) ensure that fields can be accessed and updated only using fully initialised objects.

Placeholder Declaration Typing Rules

Figure 5 contains metarules for placeholder declarations. We classify variable declarations depending on what kind of variable is used in the initialisation expression:

- **neutral** (V-DEC-N) initialisation expressions $e_1 \ldots e_n$ do not use the introduced variables $x_1 \ldots x_n$. This type of variable declaration is completely equivalent to conventional Java, but when a local variable with object type is introduced, the corresponding initialisation variable has to denote a fully initialised object. See $\mathsf{objectTypes}(\Gamma)$ in the first premise.
- **closed** (V-DEC-C) initialisation expressions $e_1 \ldots e_n$ use variables $x_1 \ldots x_n$ with placeholder types $C_1' \ \ldots \ C_n'$ and no other placeholder declared outside this specific placeholder declaration. This result is obtained by the notation ($\mathsf{objectTypes}(\Gamma)$).
  At the end of the variable declaration clauses, the introduced variables denote fully initialised objects, that is, the terminating expression can see the declared variables as their object types $C_1 \ldots C_n$.

For example, we can type check the following code:

```
class C{C myC; C(C′ myC){this.myC=myC;}}
class User{
  C makeC(){
    C x= new C(x);
    return x;
  }}
```

-FJ′-

Class `User` has a `C makeC()` method performing circular initialisation. The placeholder `x` has type `C′`. **`new`** `C(x)` is of type `C`, thanks to (NEW-T) **`new`** `C(x)` can be used to initialise variable `C x`.

– **open** (V-DEC-O) initialisation expressions $e_1 \ldots e_n$ can see **both** introduced variables $x_1 \ldots x_n$ with placeholder types $C_1' \ldots C_n'$ **and** other placeholders from enclosing placeholder declarations (thus no usage of $\mathsf{objectTypes}(\Gamma)$). This rule is applied when partially initialised variables are not guaranteed to become fully initialised, that is, the terminating expression typing must take into account the possibility that introduced variables will denote partially initialised objects until any enclosing placeholder declarations complete phase 3 initialisation.
To ensure soundness we type the terminating expression twice: once in a context where the declared variables have their object types $C_1 \ldots C_n$, and again in a context where the declared variables have their placeholder types $C_1' \ldots C_n'$. In this way the result of the expression can be an object type, but we guarantee that the resulting value is never used as a receiver.
Consider the following example:

```
class C{C myC; C(C′ myC){this.myC=myC;}}
class User{
  C makeCPart(C′ y){
    //typed with v-dec-o
    C x= new C(y);
    return x;}
  C makeCAll(){
    //typed with v-dec-c
    C z=new C(this.makeCPart(z));
    return z;}
```

-FJ′-

Method `C makeCPart(C′ y)` takes a placeholder and returns a partially initialised object. Variable `y` inside **`new`** `C(y)` is a placeholder declared outside the initialisation of variable `x`. Rule (V-DEC-C) cannot be applied. Indeed (V-DECL-C) would apply $\mathsf{objectTypes}(\Gamma)$; in this way `y` would not be in scope in expression **`new`** `C(y)`, thus **`new`** `C(y)` would be not well typed. However, (V-DEC-O) can be applied smoothly.

As you can see, the point where an object is ensured to be fully initialised is a type property, not a purely syntactical notion. This allow us to safely express initialisation patterns where the work of Syme [20] would have signaled a (false positive) dynamic error.

Memory Typing

(MEM-OK) in Figure 6 defines a well typed memory. Note how this judgement requires a

(MEM-OK)
$$\frac{}{\Gamma \vdash \mu : \text{ok}}$$
with
$\mu = \iota_1 \mapsto C_1(\overline{v}_1) \ldots \iota_n \mapsto C_n(\overline{v}_n)$
$\forall \iota \mapsto C(v_1 \ldots v_k) \in \mu :$
$p(C) = \texttt{class}\ C\ \texttt{implements\_\{}\ C_1\, f_1\texttt{;}\ \ldots\ C_k\, f_k\texttt{;}\ \overline{md}\ \texttt{\}}$
$\forall j \in 1..k : \Gamma(v_k) \leq C_k$ or $\mu(v_k) = C'_k(\_)$ and $C'_k \leq C_k$

Fig. 6. Typing rule for memory

variable environment for the placeholders. The memory is well typed if for all the objects in the memory, all fields contain either an object of the right type or a placeholder of the right type. From a well typed memory we can extract the memory environment with the following two notations:

- A memory environment typing all the objects in their corresponding object type: $\Sigma^{\mu}(\iota) = C$ iff $\mu(\iota) = C(\_)$
- A memory environment typing all fully initialised objects using their corresponding object type and all the partially initialised objects using the corresponding placeholder type:
  $\Sigma'^{\mu}(\iota) = C$ iff $\mu(\iota) = C(\_)$ and $\mathsf{reachPh}(\iota, \mu) = \emptyset$
  $\Sigma'^{\mu}(\iota) = C'$ iff $\mu(\iota) = C(\_)$ and $\mathsf{reachPh}(\iota, \mu) \neq \emptyset$
  Where $\mathsf{reachPh}(\iota, \mu)$ denotes the set of reachable placeholders.

As for rule (V-DEC-O) a well typed expression has to be typed twice: first considering fully initialised objects with object types, and second considering fully initialised object with placeholder types.

Classes and Interfaces

In Figure 7 we present standard typing rules for classes, interfaces and methods.

For any well-typed program all classes and interfaces are valid. Rule (CLASS) validates a class if all methods are valid and if the interfaces $\overline{C}$ are correctly implemented; that is, for all methods of all the implemented interfaces, a method with an analogous header is declared in the class. Note how methods are validated in the context of their class. Similarly, rule (INTERFACE) validates an interface if the interfaces $\overline{C}$ are correctly implemented; that is, for all the method headers of all the implemented interfaces, an analogous method header is declared in the interface. Finally rule (METH-T) is straightforward.

3.1 Soundness

Now we can proceed with the statement of soundness:

Theorem 1 (Soundness). *For all well typed programs p and for all expressions e under p, if $\emptyset;\emptyset;\vdash e : T$ and $\emptyset \mid e \xrightarrow{*} \mu \mid e'$, then either e' is of form ι or $\mu \mid e' \rightarrow \_ \mid \_$*

As usual, soundness can be derived from progress and subject reduction properties.

$$\boxed{\vdash cd : \text{ok} \qquad \vdash id : \text{ok}}$$

$$\text{(CLASS)}\ \frac{\forall i \in 1..n : \quad C_0 \vdash md_i : \text{ok}}{\vdash \texttt{class}\ C_0\ \texttt{implements}\ \overline{C}\texttt{\{}\ \overline{fd}\ md_1 \ldots md_n\texttt{\}} : \text{ok}}$$
with
$\forall C \in \overline{C}, m$ such that $p(C).m = T\ m\texttt{(}\overline{T\ x}\texttt{);}$,
$T\ m\texttt{(}\overline{T\ x}\texttt{)}\ \_ \in md_1 \ldots md_n$

$$\text{(INTERFACE)}\ \frac{}{\vdash \texttt{interface}\ C_0\ \texttt{extends}\ \overline{C}\texttt{\{}\ mh_1\texttt{;} \ldots mh_n\texttt{;\}} : \text{ok}}$$
with
$\forall C \in \overline{C}, m$ such that $p(C).m = T\ m\texttt{(}\overline{T\ x}\texttt{);}$,
$T\ m\texttt{(}\overline{T\ x}\texttt{);} \in mh_1 \ldots mh_n$

$$\boxed{C \vdash mh\ e : \text{ok}}$$

$$\text{(METH-T)}\ \frac{\texttt{this}{:}C, x_1{:}T_1, \ldots, x_n{:}T_n; \emptyset \vdash e : \_ \leq T}{C \vdash T\ m\texttt{(}T_1\ x_1\texttt{,} \ldots\texttt{,}\ T_n\ x_n\texttt{)}\ e : \text{ok}}$$

Fig. 7. Typing rules for classes, interfaces and methods

Theorem 2 (Progress). *For all well typed programs p and for all expressions e and memory μ under p, if $\Gamma \vdash \mu : ok$, $\mathsf{objectTypes}(\Gamma); \Sigma^{\mu} \vdash e : C$, $\Gamma; \Sigma'^{\mu} \vdash e : \_$ and $\mathsf{objectTypes}(\Gamma) = \emptyset$, then either e is of form v, or $\mu \mid e \rightarrow \_ \mid \_$*

Theorem 3 (Subject Reduction). *For all well typed programs p and for all expressions e and memory μ under p, if $\Gamma; \Sigma^{\mu} \vdash e : T$, $\Gamma \vdash \mu : ok$, $\Gamma; \Sigma'^{\mu} \vdash e : \_$ and $\mu \mid e \rightarrow \mu' \mid e'$ then $\Gamma \vdash \mu' : ok$, $\Gamma; \Sigma^{\mu'} \vdash e' : T$, $\Gamma; \Sigma'^{\mu'} \vdash e : \_$ and $T' \leq T$.*

The proofs can be found in the accompanying technical report [17].

4 Expressive Power

We show now some examples of what can be achieved with FJ′Note that these examples never use the field update operation. This allows all the examples work with immutable data structures, and it would be easy to integrate placeholders with a type system offering immutability [1,22].

Circular linked list
A clearly interesting example of expressive power is an arbitrarily sized, immutable, circular list. Note how `ListProducer` can be a user class, and does not need to know implementation details of the `List`. (Other approaches [19] would require the production process to happen inside `List` class, encoded in the constructor.)

```
class List{
  final int e; final List next;
  List(int e, List' next){this.e=e; this.next=next;}
  }
class ListProducer{
  List' mkAll(int e, int n, List' head){
    if(n==1) return new List(e, head);
    return new List(e, this.mkAll(e+1, n-1, head));
    }
  List make(int e,int n){
    List x = this.mkAll(e, n, x);
    return x;
    }
  }
...
  new ListProducer().make(100,10)
```

A circular list `List` has a field `e` containing a value and a field `next` containing a `List`.

Method `mkAll` takes three parameters: a value `e`, a length `n` and a list `head`. Method `mkAll` creates a list of length `n` containing values starting from `e` as list elements, ending with list `head`. Finally, method `make` takes a value `e`, a length `n` and creates a circular list of length `n`. Method `make` performs the circular initialisation and returns a fully initialised `List`. Note how `make(100,10)` can be directly used to make a circular list of numbers 100,101,...109.

Doubly linked list

We can convert the singly linked list example to implement an immutable doubly linked list:

```
class List{
  final int e; final List next; final List pred;
  List(int e, List' next,List' pred){
    this.e=e; this.next=next;this.pred=pred;
    }
  }
class ListProducer{
  List' mkAll(int e, int n, List' pred){
    if(n==1){
      List x=new List(e, x, pred);
      return x;
      }
    List x=new List(e, mkAll(e+1,n-1,x), pred);
    return x;
    }
  List make(int e,int n){
    List x = this.mkAll(e, n, x);
    return x;
    }
}
...
  new ListProducer().make(100,10)
```

Here, the produced list is a (non circular) doubly linked list, where termination is represented by having the same value for **this** and **this**.next or **this**.pred. If our List was mutable, we could make it circular after creation. To the best of our knowledge, in the type system presented in this paper, it is impossible to create an immutable doubly-linked list: we would need either multiple return values for a method or support for fields with placeholder types. A more complex version of FJ′, supporting placeholder fields, can be found in an associated technical report [18].

Parser Combinators

As Gilad Bracha suggests [2], it is possible to leverage the recursive nature of parsers in order to define classes that represent different typical operations in a BNF grammar. With overloading support for operators (|) and (,) (as in C++ or Newspeak) it is possible to obtain a syntax very near to conventional BNF. For example we could obtain the following:

parser combinator

```
Production operator|(Production left, Production right){
  return new OrProduction(left,right);}
Production operator,(Production left, Production right){
  return new SeqProduction(left,right);}
...
  Production number= term(1,9)| term(1,9), number0,
  Production number0= term(0,9) | term(0,9), number0,
  Production e= number | e,term("*"),e | e,term("+"),e;
```

where term is a static method. Thanks to placeholders, we can use number0 and e recursively.

The current implementation of parser combinators in Newspeak solves this problem in a much more ad-hoc solution, using reflection [2]. Designed concurrently with our placeholders, Newspeak 0.08 introduced "simultaneous slot definitions" that use futures:

" A simultaneous slot declaration with a right hand side expression e initialises the slot to the value of p computing: e, where p is the class Past'Future. The result is a future that will compute the expression e on demand. All these futures are resolved once the last slot declaration in the simultaneous slot definition clause has been executed. Past'Future implements a pipelined promise so that any well founded mutual recursion between simultaneous slots will resolve properly. "

Futures in Newspeak are objects forwarding all messages to the result of the computation. In this way, Newspeak can provide a similar expressive power to placeholders. Thanks to the dynamic nature of Newspeak, there is no type guarantee of well formedness of a circular initialisation using Newspeak futures. Newspeak requires an extra level of indirection (even if transparent in most of the cases), and the execution order of the different initialisation expression is "on demand" instead of sequential.

5 Implications for Implementation

In this section we discuss some options that could be used to implement placeholders.

Smalltalk offers a method called "become:" that changes object references. After "a become: b" all local variables and all object fields originally referring to the object

denoted by `a`, now refer to the object that was denoted by `b`, and vice versa. An implementation of placeholders over a virtual machine offering a "`become:`" method is very simple; for example, our initial placeholder declaration could be written as follows:

translation with become:

```
//placeholders initialisation
Security sP=new Security(null);
DBA dbaP=new DBA(null,null);
SMS sms=new SMS(null,null);
GUI guiP=new GUI(null,null,null);
//real initialisation
Security s=new Security(dbaP);
DBA dba=new DBA(sP,guiP);
SMS sms=new SMS(sP,dbaP);
GUI gui=new GUI(smsP,dbaP,sP);
//placeholders replacement
sP.become(s);  dbaP.become(dba);
smsP.become(sms);  guiP.become(gui);
```

Is it possible to emulate "`become:`" on a platform that does not support it natively? Of course, a general purpose "`become:`" comes with the prohibitive cost of the full heap scan; however the restricted usage of "`become:`" in our case can be implemented efficiently even in Java. The main idea is to produce a placeholder subclass (interface `Ph`) of each class that could be used as a placeholder, and to make each class that can be initialized using a placeholder implement the `ReplacePh` interface that defines a method to replace placeholders stored in fields with the placeholders' actual objects.

In this scheme, whenever an object is allocated using placeholders, we notify those placeholders that the newly allocated object refers to them. When a placeholder declaration completes initialisation, the introduced placeholders can be correctly and efficiently replaced. In the detail, such translation would:

- generate interfaces `ReplacePh`, `Ph` and class `PhAdd`

-Java-

```
interface ReplacePh{void replacePh(Ph ph, Object o);}
interface Ph{ List<ReplacePh> getList();}
class PhAdd{
public static<T> T add(T fresh, Object ... phs){
  for (Object o:phs)
    if(o instanceof Ph)
      ((Ph)o).getList().add((ReplacePh)fresh);
  return fresh;
  }}
```

  `ReplacePh` represents an instance whose fields can contain a placeholder `ph`, that can be replaced with `o` when the object is available;; `Ph` represents a placeholder, and can provide the list of all the objects whose fields point to that placeholder; `PhAdd.add` notifies placeholders in `phs` that the `fresh` object contains them in one of its fields.
- map both placeholder types and object types to the corresponding simple Java type; that is all the types of form C and C'' will be mapped to C.

– generate for all the classes and interfaces a "placeholder" class, extending the original one and the `Ph` interface, providing a no arguments constructor and a list of `Objects` containing all the objects whose fields refer to this "placeholder object",
– makes every class originally present in FJ′ implement `ReplacePh`. For example

FJ′
```
interface T1{...}
class C implements T1{
  T1 x; T2 y; C(T1' x,T2' y){this.x=x; this.y=y;}
  ...}
```

would be translated into

Java
```
interface T1{...}
class PhT1 implements T1,Ph{
  List<ReplacePh> list=new ArrayList<ReplacePh>();
  List<ReplacePh> getList(){return this.list;}
  .../*any method of T1 throws Error*/}

class C implements T1,ReplacePh{
  T1 x; T2 y; C(T1 x,T2 y){this.x=x; this.y=y;}
  void replacePh(Ph ph, Object o){
    if(this.x==ph)this.x=(T1)o;
    if(this.y==ph)this.y=(T2)o;
    }
  ...}
class PhC extends C implements Ph{
  List<ReplacePh> list=new ArrayList<ReplacePh>();
  List<ReplacePh> getList(){return this.list;}
  .../*any method of C throws Error*/}
```

– translate placeholder declarations into the three phase initialisation, so that:

FJ′
```
T1 x= new C().m(x,y),
T2 y= new D().m(x,y);
new K().m(x,y);
```

would be translated into

Java
```
//(1)initialise dummy placeholder objects
T1 _x= new PhT1();
T2 _y= new PhT2();
//(2) run normal initialisation code, using placeholder
T1 x= new C().m(_x,_y),
T2 y= new D().m(_x,_y);
//(3) replace placeholders with the correct value
for(ReplacePh o:_x.getList())  o.replacePh(_x,x);
for(ReplacePh o:_y.getList())  o.replacePh(_y,y);
new K().m(x,y);
```

– finally, for any constructor parameter that is statically a placeholder, we use static method `PhAdd.add` to insert the newly created object into the placeholder list; for example

FJ′
```
class C implements T1{
T1 m(T1′ x,T2′ y){ return new C(x,y);}
...}
```

would be translated into

Java
```
class C implements T1,ReplacePh{
T1 m(T1 x,T2 y){ return PhAdd.add(new C(x,y),x,y);}
...}
```

6 Related Work

Not embracing any sort of laziness
Many approaches in the area of circular initialisation do not support any form of laziness, and thus cannot support the initialisation example from our introduction in the obvious way. The construction process of one entity is interleaved with the initialisation process of other entities and they rely on explicit mutation to create circular object graphs.

Hardhat Constructors [8,21] restrict constructors to avoid the leak of partially initialised objects out of the constructor scope. Similar techniques can be employed by FJ′ to extend our calculus with more expressive constructors. OIGJ [23] and others [9,10] allow the creation of *immutable* object graphs by using the concept of "commitment points" (where the mutable object graph can be *promoted* to immutable). These approaches offer no guarantee against null pointer exceptions.

As Manuel Fahndrich, Songtao Xia, and Don Syme have astutely observed [5,20], recursive bindings in a functional language like OCaml [14] serve a purpose very similar to placeholders:

OCaml
```
type t = A of t * t | B of t * t
let rec x = A( x, y )
  and y = B( y, x )
```

Here, two variables (`x` and `y`) are circularly initialised using the **`let rec`** recursive binding expression. OCaml imposes heavy restrictions [5]: *" the right-hand side of recursive value definitions to be constructors or tuples, and all occurrences of the defined names must appear only as constructor or tuple arguments. "*
These restrictions rule out any form of laziness, and makes it impossible to express any variations of the Factory pattern [7] as method calls are prohibited in such initialisation expressions. However, thanks to these restrictions, the following implementation for such recursive bindings is possible: first allocate memory for the values being constructed — once all the bindings are established, the constructed values can be initialised normally.

Delayed Types [5] lift those limitations, but still are unable to provide good support for factory methods: indeed Delayed Types require factory methods to expose fields in their type annotations. In personal communication, Fahndrich agrees that this would break encapsulation for private fields and is in general not feasible in case of interfaces as return types. Moreover, both Delayed Types [5] and Masked types [16] require an explicit two step initialisation, where the values are created and the circular references are fixed explicitly afterward; that is, these two works try to verify code similar to the following:

```
-Java-  class C{ C myC; C(C myC){this.myC=myC;} }
        ... C c=new C(null);
            c.myC=c;
```

This kind of code is hard to maintain: when a modification to the internal implementation is needed, all the code that initialises new instances can be broken. Delayed Types and Masked Types [5,16] ensure the absence of `NullPointerException`s in this kind of code.

Freedom Before Commitment [19] proposes another approach, where constructors trigger the construction process of their sub-components:

```
-Java-  class A{ B myB; A() { myB=new B(this);}}
        class B{ A myA; B(A x){ myA=x;}}//here x.myB==null
```

The code inside the constructor of `B` cannot freely manipulate the parameter `x`, indeed if the control flow starts from the expression **`new`** `A()`, the field `x.myB` is still **`null`** when variable `x` is visible. In [19] the absence of `NullPointerException`s is proved. This avoids an explicit two step initialisation, but leverages a sequence of recursive constructor calls where one of them triggers the initialisation of the other components.

To conclude, these works [5,16,19] focus on statically preventing null pointer exceptions, supporting safe initialisation in OO languages as they are at the moment. Our goal is to change the language semantics to support the intuitive idea that the client code is responsible for knowing the dependencies between the different components in the system, so that factory methods can receive correct parameters.

From the type system point of view we share many similarities to Summers and Müller's Freedom Before Commitment. This system has partially initialised objects (called free), like FJ′, although we do not need to expose an explicit *partially initialised object type*. Unlike FJ′, Freedom Before Commitment relies on constructors, and does not support factory methods. Summers and Müller have conducted an extensive study of applicability of their approach, and they do consider factories instead of constructors. Replacing constructors with factory/creation methods is suggested by Fowler [6] and is empirically shown to be a good methodology [4].

Uses some form of laziness

Languages with lazy semantics, like Haskell, can use laziness/thunks to encode placeholders, thus reaching an expressive power similar to FJ′; but without the static guarantees offered by our type system (this style of programming is often called *tying the knot* in the Haskell community). In FJ′, dereferencing a placeholder is a dynamic error similar to a `NullPointerException` (a stuck state in the formal model) while in Haskell, accessing a thunk performs the associated computation on demand.
FJ′ and Haskell also differentiate when placeholders are replaced or thunks evaluated:

- In Haskell a thunk is evaluated when its value is needed and, as soon as the computation ends, all the occurrences of the thunk are replaced with the corresponding value. That is, the lifetime of a thunk is unbounded, and in some cases it is possible for a thunk to not be evaluated at all, saving computation time.

– In FJ′ initialisation expressions are executed in the conventional top down, left to right order; placeholders introduced in a placeholder declaration are replaced as soon as the semicolon is reached. The lifetime of a placeholder is lexically bounded: when the initialisation is concluded the placeholder is replaced. Placeholders retain the simplicity and predictability of the call by value semantics.

Both in Haskell and in FJ′ a term like `T x = x;` is meaningless; those degenerate cases are not so uncommon as one can suppose. In Haskell, a non trivially degenerate case is the following:

Haskell
```
data L = Lnil | Lcons Int L
head :: L->Int
head Lnil=0
head (Lcons n a) = n
main=(let x=x in putStr(show(head x)))
```

Here type `L` denotes a list of integers, with the two conventional type constructors. The function `head` returns the head of the list, or `0` for the empty list. It is clear that the definition of the `main` makes no sense: we call it a *non well-guarded definition*.

In order to explain the importance of this problem, we now show an example where a non well-guarded expression emerges from an apparently benign code. Indeed in the general case is very difficult to spot non well-guarded definitions.
Consider the following two functions `f1` and `f2`:

Haskell
```
f1 :: L -> L                    f2 :: L -> L
f1 a= (LCons 42 a)              f2 a= a
```

Function `main1` correctly initialises an infinite list containing only the number 42, and shows 42. However, function `main2` is a non well-guarded definition:

Haskell
```
main1=(let x=(f1 x) in putStr(show(head x)))
main2=(let x=(f2 x) in putStr(show(head x)))
```

Since the only difference is the occurrence of `f2` instead of `f1`, you can see that in Haskell, without knowing the code of `f1` and `f2`, it is impossible to predict whether some code is well-guarded or not. In fact, executing `main1` in Haskell results in printing `42` while executing `main2` results in an infinite loop.

In the following example we show the code of the last example rephrased in FJ′; encoding functions as methods on classes. `User1` is verified by our type system, while `User2` is ill-typed. Note how classes `User1` and `User2` differ only in the implementation of method `f`. `User1.main` produces an infinite list composed by a single cell with value 42, while `User2.main` is ill typed.

FJ′
```
class List{ int e; List next; }
class User1 {
  List f(List' x){ return new List(42,x);}
  List main(){ List x=f(x); return x; } } //safe
class User2{
  List f(List' x){ return x; }
  List main(){ List x=f(x); return x; } } //unsafe
```

The significance of this problem is highlighted in a proposal by Syme [20]. He uses a disciplined form of laziness in order to design a language which is very similar in spirit to our approach, with roughly as much power but an additional requirement for dynamic checks in order to avoid non well-guarded cases to be evaluated in the context of data recursion, i.e. circular initialisation.

7 Conclusion

"I call it my billion-dollar mistake. It was the invention of the null reference"

Tony Hoare, "Null References: The Billion Dollar Mistake" [12]

Today, writing correct and maintainable code involving circular initialisation is very difficult: most practical solutions rely on null values — Tony Hoare's "Billion Dollar Mistake". This is an important problem since many software architectures and natural phenomena involve circular dependencies. As in the "chicken and egg" paradox, circular dependencies are hard for humans to understand and to reason about. This has hampered the development of effective techniques for supporting safe circular initialisation in software engineering, and their support in programming languages.

There have been many proposals to solve the problem of circular data structure initialisation [19,14,5,16]. In our opinion FJ′ offers a simpler type system compared to these approaches; moreover it allows simpler and cleaner programming patterns to be used. We can obtain such simplicity because we attack the problem from two different angles: first we define a new concept — placeholders — with intuitive semantics, and then we develop a type system to ensure placeholders are used safely. We hope this gives placeholders a good chance of adoption by real language implementations.

We conclude with another well known quote by Hoare:

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously *no deficiencies, and the other is to make it so complicated that there are no* obvious *deficiencies. The first method is far more difficult."*

Tony Hoare, "The Emperor's Old Clothes" [11]

We have tried our best to follow the first way, and the result looks pretty simple to us. Whether is it simple enough to fix the billion dollar mistake, only time will tell.

Acknowledgments. Thanks to Gilad Bracha, Alex Summers, Peter Müller, Manuel Fahndrich, David Pearce, Elena Zucca and Jonathan Aldrich. This work is funded by RSNZ Marsden Fund.

References

1. Birka, A., Ernst, M.D.: A practical type system and language for reference immutability. In: OOPSLA, pp. 35–49 (2004)
2. Bracha, G.: Executable grammars in Newspeak. In: JAOO (2007)
3. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in Newspeak. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 405–428. Springer, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1883978.1884007
4. Counsell, S., Loizou, G., Najjar, R.: Evaluation of the 'replace constructors with creation methods' refactoring in Java systems. IET Software 4(5), 318–333 (2010)
5. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA. pp. 337–350 (2007)
6. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2000)
7. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley (1995)
8. 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)
9. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. In: OOPSLA, pp. 21–40 (2012)
10. Haack, C., Poll, E.: Type-based object immutability with flexible initialization. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 520–545. Springer, Heidelberg (2009)
11. Hoare, C.A.R.: The emperor's old clothes. Comm. ACM 24(2) (February 1981)
12. Hoare, C.: Null references: The billion dollar mistake (March 2009), abstract of QCon London Keynote qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake
13. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: OOPSLA, pp. 132–146 (1999)
14. Leroy, X.: The Objective Caml system (release 2.00) (August 1998), http://paulliac.inria.fr/caml
15. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses - practical extensibility with object algebras. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 2–27. Springer, Heidelberg (2012)
16. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL, pp. 53–65 (2009)
17. Servetto, M., Mackay, J., Potanin, A., Noble, J.: The billion dollar fix: Safe modular circular initialisation with placeholders and placeholder types. Tech. Rep. 12-25, ECS, VUW (2012), http://ecs.victoria.ac.nz/Main/TechnicalReportSeries
18. Servetto, M., Potanin, A.: Our billion dollar fix. Tech. Rep. 12-19, ECS, VUW (2012), http://ecs.victoria.ac.nz/Main/TechnicalReportSeries
19. Summers, A.J., Müller, P.: Freedom before commitment - a lightweight type system for object initialisation. In: OOPSLA, pp. 1013–1032 (2011)
20. Syme, D.: Initializing mutually referential abstract objects: The value recursion challenge. Electronic Notes in Theoretical Computer Science 148(2), 3–25 (2006)
21. Zibin, Y., Cunningham, D., Peshansky, I., Saraswat, V.: Object initialization in X10. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 207–231. Springer, Heidelberg (2012)
22. Zibin, Y., Potanin, A., Artzi, S., Kiezun, A., Ernst, M.D.: Object and reference immutability using Java generics. In: Foundations of Software Engineering (2007)
23. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in generic Java. In: OOPSLA, pp. 598–617 (2010)

Implementing Federated Object Systems*

Tobias Freudenreich[2], Patrick Eugster[1], Sebastian Frischbier[2], Stefan Appel[2], and Alejandro Buchmann[2]

[1] Department of Computer Science, Purdue University, USA
p@cs.purdue.edu
[2] Databases and Distributed Systems Group (DVS), TU Darmstadt, Germany
{lastname}@dvs.tu-darmstadt.de

Abstract. With the increase of automatically sensed and generated data in distributed software systems, the publish/subscribe paradigm gains importance. Automatically generated notifications are pushed from their publishers to interested subscribers. Interoperability is a core issue in such federated networked distributed applications. However, the problems of heterogeneity must be reconsidered in the light of tougher conditions than previously: low latency delivery in addition to expressiveness and extensibility.

To aid in engineering federated distributed systems, this paper proposes a framework for object transformations. Components can operate in individual, semantic contexts, which include local type declarations, fine-grained transformation rules (t-rules), and type mappings that express the programmer's intent at a high level. Our generic approach supports transformations at any granularity using clear priorities to select among complementary t-rules.

We present empirical evidence of the efficiency of our approach and of the benefits to the programmer in terms of code quality.

Keywords: Heterogeneity, Publish/Subscribe, Semantic Decoupling, Transformations, Events.

1 Introduction

In today's distributed software systems, vast amounts of data are generated and processed automatically, such as the tracking of goods by continuous updates called *event notifications*. Due to the high frequency of such data, distribution to clients must happen automatically based on client interests. This makes *implicit invocations* [32,39] (publish/subscribe – pub/sub [29]) the paradigm of choice. In these systems, subscribers express their interests in data in the form

* Funded in part by US NSF grants # 0644013 and # 0834529, DARPA grant # N11AP20014, Alexander von Humboldt foundation, and the German Federal Ministry of Education and Research (BMBF) grants # 01IC10S01 and # 01IS12054. This work was performed within the LOEWE Priority Program Dynamo PLV (http://www.dynamo-plv.de) supported by the LOEWE research initiative of the state of Hesse/Germany. The authors assume responsibility for the content.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 230–254, 2013.

of subscriptions. Broker nodes route matching notifications from publishers to the appropriate subscribers. Communication thus happens in an *n-to-m* fashion without direct references between communicating components. The federated software systems supporting these communications are loosely-coupled, highly heterogeneous and developed by many parties.

Heterogeneity Is Approaching. The need for such systems becomes apparent in today's economy, where large-scale software systems manage complex supply-chain networks. Cooperation happens across the globe between companies of different countries, cultures and structures. Software systems generate a huge amount of information that has to be distributed in business real-time among software components in a flexible way. For example, today's complex supply chains involve many companies world-wide; production strategies like *just-in-time production* have strongly increased the need for continuous low-latency flows of information between participants in a chain (e.g., monitoring transported goods) [13].

The continuing globalization of the economy[1], along with disruptive trends like the *Internet of Things* or *ubiquitous computing resources* promoted by the cloud paradigm and availability of event notification systems for clouds (e.g., Amazon Simple Notification Service [1]), will lead to more heterogeneous push-based distributed systems [20]. Similarly, the proliferation of social networks interconnecting people with different cultural backgrounds and the use of notification services for communication therein (e.g., LinkedIn's Kafka [7]) further supports the trend.

Integration Is Challenging. The problem that arises in such heterogeneous environments are different data representations and data semantics of components. Local interpretations – *contexts* – differ based on geographical, cultural, legal, but also technical reasons. Programming languages differ in their notions of types, and even within a language, two sets of modules developed independently are not always easily integrated, e.g., due to *single* inheritance [12]. However, without correct interpretation of data, proper matching of *event objects* to subscriptions will fail as the anonymous n-to-m interaction between the components does not reveal intended bindings. Integration of information flows between publishers and subscribers is challenging as it must fulfill a number of requirements:

Expressiveness. Mediating between different unit systems (e.g., Fahrenheit and Celsius) requires value-based transformations. Many entities are encoded with several attributes, without 1:1 correspondances between types (e.g., Cartesian and Polar coordinates), and certain integrations might involve adding attributes (e.g., adding an attribute with a default or **null** value such as the state for a European surface mail address in US format).

[1] Our two ongoing research projects DynamoPLV(http://www.dynamo-plv.de) and EMERGENT(http://www.software-cluster.org) investigate such scenarios.

Efficiency. Given the high rates at which event objects are published, mediation must take place on the fly with low latency.

Adaptability. The considered systems need to be able to accomodate joining and leaving of client components. Such changes can also engender new kinds of integrated data. Given the size of these systems, stopping them to add new components or also to modify existing integration rules is infeasible.

In addition, it should of course be easy for a programmer to express how integration has to happen. Existing approaches do not address all of these requirements to the full extent, as detailed in Section 7.

Transformation Supports Integration. In this paper we propose a framework for programming federated distributed software centered on object *transformations*. In our approach, we assume that each component (i.e., subscriber, publisher or pub/sub broker) resides in its own semantic context that involves a set of (abstract) parameters (e.g., country, programming language, development team) which governs how the component interprets event objects. We advocate a facility to define contexts including (1) *local types*, (2) fine-grained declarative *transformation rules* (*t-rules*) to specify the desired transformations between types, and (3) high-level *type mappings* expressing the programmer's intent and used to verify that the t-rule application yields consistent outcomes. Figure 1 shows an overview of our approach. Contexts can be extended at runtime and reused across components.

Our intermediary model avoids defining every possible transformation from any publisher to any subscriber, in a way similar to the Canonical Data Model in Enterprise Application Integration [22]. Thus we avoid $n \times m$ complexity of the transformation set and keep the set maintainable. This does not sacrifice adaptability however, since we do not require that the Canonical Data Model is established a-priori. Instead, it can be extended and modified at runtime by defining new contexts and compiling rule-sets on the fly. In fact, an intermediary model decouples publishers and subscribers further in that changes to a publisher's context (i.e. its transformation rules) do not affect its subscribers. Our model does not make assumptions on the specification language and is thus language-agnostic.

Contributions and Roadmap. In this paper we

1. present an expressive and extensible model for federated software systems targeting mainstream programming languages used for such applications (e.g., C++, C#, Java) centered on *transformation rules* (t-rules);
2. introduce *contexts* as a reusable, higher-level abstractions of transformations, providing easy maintainability;
3. describe the implementation of our model in the ACTReSS system [19] based on the popular open-source ActiveMQ [37] event broker;
4. evaluate our implementation. We demonstrate how extracting and consolidating transformation code improves efficiency compared to an equally

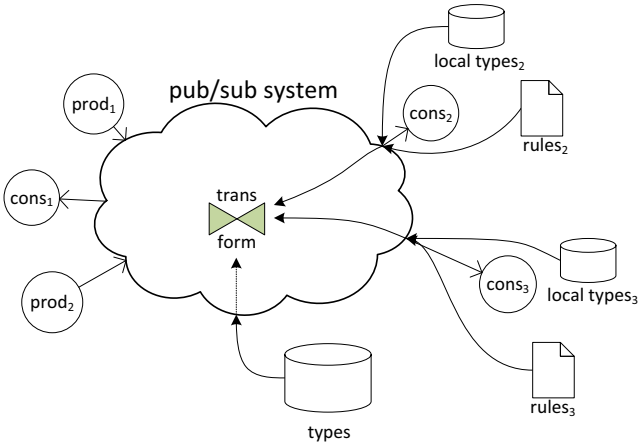


Fig. 1. Architectural overview. Producers ($prod_i$) send messages to consumers ($cons_j$) through a pub/sub system. Each client supplies a set of mapping rules between the global types and its local types (local types and rules omitted for some clients for presentation purposes).

adaptable and expressive approach without inherent support, i.e., based on reflection, and equal efficiency to manually coded transformations at client components; we also show that code quality, in particular upon adaptation, is much improved compared to manual coding or monolithic object transformations based on existing frameworks.

In a companion report [17] we formalize a subset of our model supporting only single subtyping and prove its soundness. An earlier implementation of our ACTrESS prototype was presented in a previous publication [19] without breaking down contexts into t-rules and type mappings – the main artifacts that a system software engineer must deal with – and detailing their syntax and semantics. Evaluation elided costs for extension or code quality improvements.

The remainder of this paper is organized as follows. Section 2 presents preliminary information. Section 3 presents t-rules through intuitive examples. Section 4 details contexts with particular focus on their relationships and type mappings. Section 5 presents the implementation of our model in ACTrESS. Section 6 evaluates its benefits. Section 7 discusses related work and Section 8 draws conclusions.

2 Preliminaries

We outline the notions of objects and transformations considered in this paper, using Figure 2 for illustration. Please note that for presentation purposes, the given example is small; our model allows for more complex transformations.

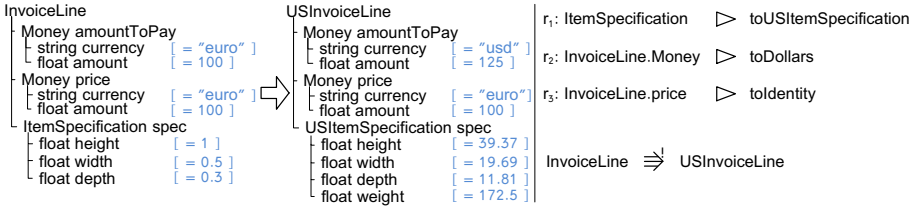


Fig. 2. Sample transformation. InvoiceLine and USInvoiceLine are context-specific interpretations of same objects. Note that currency is typically encoded explicitly, but units for other values are almost never encoded. Simplified transformation rules (t-rules) and type mappings are illustrated on the right.

2.1 Objects

We consider event objects in the general form of typed records of attributes. More precisely, such an object is an instance of a (complex) *type* ($\overline{z}$ denotes a sequence $z_1 \ldots z_n$):

Definition 1 (Type). *A type T is either a primitive type or a complex type declared as T* **extends** *$T_1,\ldots,T_n[a_1:T'_1,\ldots,a_w:T'_w]$. $\overline{T}$ are super-types (complex) of T. T's attributes include those of all its super-types as well as $\overline{a}$.*

Thus, in a nested fashion, attributes of event objects can be objects. For example, InvoiceLine $[\overline{\delta}]$ would represent an event object of type InvoiceLine defined in Figure 2. The record $[\overline{\delta}]$ contains a sequence of objects corresponding to the attributes of InvoiceLine (i.e., amountToPay, price, spec) which are of respective types defined by InvoiceLine (Money, Money, and ItemSpecification). We consider all transferred objects to be *values*, and thus our considerations also apply to other remote communication models with value semantics (references are usually built atop [5]). Note that an event object is an object but not every object is an event object.

We omit member functions/methods from types as our approach does not require those to be defined as part of types. The function $attrs(T)$ returns the attributes $\overline{a}$ of type T with their respective types $\overline{T}$. $T \preceq T'$ means that T is a subtype of T', i.e., T has been explicitly declared as a subtype of T', or one of the super-types of $T$is a subtype of T'. Our approach does not depend on the way in which attribute name clashes are handled.

2.2 Transformations: An Intuition

A very simple form of transformation of objects consists in modifying values of attributes which are of specific primitive types such as **float**s. These attributes can have a unit associated with them, e.g., meters or inches (cf. height, width and depth of InvoiceLine.spec in Figure 2). A similar case are conversions of primitive values between different architectures or platforms. On the other end of the spectrum of transformations, an object may be transformed in a way which affects

its internal structure, e.g., by merging multiple attributes, dropping attributes, and instantiating new ones (cf. weight of USInvoiceLine.spec in Figure 2). Any combination of these may be used to deal with versioning – by adding version numbers to type names and describing corresponding transformations.

There are different dimensions along which one can divide the space of object transformations. Section 7 further dissects this space in order to relate existing work to our proposed approach. The goal in our present work is to support (a) fine *granularity* – transformations on any attributes, at any nesting level, in objects; (b) strong *completeness* – function-based stateful transformations. (a) and (b) together yield the required expressiveness (see Section 1), while efficieny is supported by a decentralized application of transformations. These features as well as adaptability and ease of use are achieved by our design outlined in the following sections.

3 Transformation Rules

We introduce our approach to contextualization by starting from transformations which currently are dealt with in a very explicit manner by programmers. We follow the example of a logistics provider operating world-wide who has to communicate with customers from different countries.

3.1 Overview

Our model is centered around declarative *transformation rules* (t-rules) which minimize the necessary specifications and align well with programmers' mental models. These t-rules apply a given transformation function to attributes in event objects. The application of these transformations proceeds in a top-down fashion following the nested structure of event objects. Conceptually speaking, in example from Figure 2, imagine an object of type InvoiceLine being traversed attribute-wise, i.e., amountToPay, price, and finally spec, and any corresponding resolved t-rules being applied. If Money is a composite type, then the attributes of amountToPay are traversed recursively (depth) before proceeding with price (width). At any point in the transformation process, we thus consider one *path*:

Definition 2 (Fully qualified path). *A fully qualified path is a 2-tuple* $\langle T_0 \cdot \ldots \cdot T_n, a_1 \cdot \ldots \cdot a_n \rangle$ *such that* $\forall i \in [0..n-1]$, $attrs(T_i) = \langle \ldots a_{i+1} \ldots, \ldots T'_{i+1} \ldots \rangle$ *and* $T_{i+1} \preceq T'_{i+1}$.

A (fully qualified) path thus unambiguously and correctly denotes a given attribute within event objects. A prefix of a path (e.g., $\langle T_0 \cdot T_1, a_1 \rangle$ for $\langle T_0 \cdot T_1 \cdot T_2, a_1 \cdot a_2 \rangle$) is a path itself. Next we elaborate on how to describe t-rules and how they apply to paths.

3.2 Separating Patterns and Functions

The logistics provider from our example receives the specifications for the items to be transported from its customers. However, specifications do not have a standard format and even simple things like units vary between countries or even individual customers.

Intuitively, we would like the ability to define "default" transformations for certain types. Suppose a software service calculating the price for transporting goods. As part of the calculation, it processes instances of ItemSpecification which contain information in a specific combination of units (e.g. meters for height, width, and depth). After the calculation, the logistics provider sends the price along with these specifications to its customers. A US customer needs these properties in Imperial units (e.g., inches) rather than the logistics provider's format. A t-rule to transform all such attributes in all event objects could be simply expressed as (cf. r_1 in Figure 2):

```
ItemSpecification  ▷ toUSSpecification;
```

t-rules are thus of the following form

Definition 3 (Transformation rule). *A transformation rule (t-rule) is of the form $p \triangleright f$ where p is a* pattern *delineating a set of attributes in event type(s) and f refers to a function.*

Functions are used to transform at any path which the pattern applies to, and are defined separately from t-rules. This separation between patterns and functions is key to expressiveness and ease of use. In an object-oriented programming language such as Java, functions are typically methods on transformed objects (e.g., $p \triangleright$ m with m an instance method in the type expected from p) or **static** methods (e.g., $p \triangleright$ C.m). In the example above toUSSpecification refers to a function which takes an instance of ItemSpecification as its argument and produces an instance of an analogous type USItemSpecification, which is local to the present context:

```
USItemSpecification  toUSSpecification ( ItemSpecification  is )
    { return new USItemSpecification (...);  }
```

We will unveil the details of patterns as we move on and present a precise definition after that.

3.3 Types and Nesting

For convenience we let a pattern like the above apply *at any nesting level* within a type. Specifically, the pattern applies to any attribute of type ItemSpecification at any depth in event objects of any type.

There are cases where we want to leave certain attributes in *specific* event types unchanged (or apply a different transformation function). For example, customs declaration papers require the same units as the logistics provider usually uses. In this case, the logistics provider does not want to transform the

item specification, but just use it as it is. To achieve this, we can qualify more precisely where to apply this sort of transformation. The following t-rule

```
CustomsDeclaration.ItemSpecification ▷ toIdentity;
```

would apply an identity function (omitted here for brevity) to attributes of type ItemSpecification in CustomsDeclarations.

To overcome the cumbersome task of enumerating all event types which contain attributes of type ItemSpecification, we introduce priorities among rules. If we combine both t-rules above into one t-rule set, the second t-rule overrides the first one for CustomsDeclaration events. All other attributes of type ItemSpecification are transformed according to t-rule r_1 as depicted in Figure 2. This conveys the first intuition underlying our design: rules with more specific patterns override rules with less specific ones. In the above example, CustomsDeclaration.ItemSpecification overrides ItemSpecification. Specificity here translates to length, or, in other terms, *nesting level*.

As mentioned we consider patterns ending in types to apply to all occurrences of that type in deeper nesting levels. That is, one can picture InvoiceLine.Money as representing InvoiceLine.∗Money where ∗ can match any (possibly empty) infix. This seems natural when looking at the generic transformations specified in the preceding examples. We apply this variable nesting level only at the *last* type in a pattern though, prohibiting something like TopLevelType.∗LevelX.∗LevelY. More expressive patterns could be envisioned by removing this constraint but this is likely to come at a substantial cost in terms of simplicity for the programmer.

3.4 Attributes

Transforming by type only is sometimes too general. Consider the type InvoiceLine of Figure 2, which represents a single line on an invoice and contains the item's specifications and the cost for delivery (with its own Money type). Assume that the logistics provider needs this amount in its currency for internal bookkeeping. However, customers want this amount in their respective local currencies. Thus we cannot treat every attribute of type Money the same way.

We consequently support references to *attributes* as another element in patterns for t-rule application. The t-rule

```
InvoiceLine.amountToPay ▷ toDollars;
```

applies the following function

```
Money toDollars(Money m) {
    if (m.getCurrency() == "euro") {
        m.setCurrency("usd");
        // getRate() might depend on the current time and other state
        m.setAmount(m.getAmount() * getRate("euro","usd"));
    } else if (m.getCurrency() == "yen") {...}
    ...
    return m;
}
```

to *only* the amountToPay attribute of events of type InvoiceLine. Other attributes of type Money in an InvoiceLine or any other type are left unchanged (cf. Figure 2). Even if *succeeded* by a less specific t-rule which mandates that all Money attributes be transformed to include taxes, such as in the t-rule set

```
InvoiceLine.amountToPay ▷ toDollars;
Money ▷ addTaxToMoney;
```

attribute amountToPay would be transformed via toDollars and not addTaxToMoney.

We believe that this is a much more natural semantics than declaration order alone. The reasoning behind this second form of precedence consists in prioritizing *instances over types* on otherwise comparable patterns, i.e., attribute-level declarations over type-level declarations. The outcome above would be identical if the second t-rule used the pattern InvoiceLine.Money to define a default translation of all money attributes in type InvoiceLine.

3.5 Subtyping

Subtyping is a fundamental concept in programming languages. With nominal subtyping, in our model, any event object can at any nesting level have an attribute a carrying an instance of a type T' which is a subtype of a's declared type T ($T' \preceq_P T$). With that in mind, it seems natural to allow the programmer to define t-rules which are subtype-sensitive. For instance, we might define a type WeightedItemSpec which extends ItemSpec by adding an attribute weight of type **float**, and define a corresponding *specific* rule

```
InvoiceLine.spec(WeightedItemSpec) ▷ ...;
```

Among a set of alternative patterns it seems natural to follow *subtyping level*, i.e, pick the one which refers to the most *derived* type at a point of comparison. For example we select the above rule over one with pattern InvoiceLine.spec(ItemSpecification),or InvoiceLine.spec for short, for an attribute weight of *dynamic* type WeightedItemSpec in an InvoiceLine. Note though that, unlike in dynamic method dispatching [10] this selection itself is not done dynamically; as we will elaborate on in Section 5, relevant t-rules are resolved before actually being applied.

In conclusion from the above, we can now proceed to more formally specifying the shape of *patterns:*

Definition 4 (Pattern). *A pattern $p = T.q_1.\ldots.q_n$ consists in a type reference T followed by a (possibly empty) sequence of* qualifiers*, and denotes attributes in event objects. A qualifier refers either to all attributes with a given type (*type qualifier T*) or to a given attribute "a" with a given type T (*attribute qualifier $a(T)$*).*

We only consider *valid* patterns. That is, for any prefix $T.q_1.\ldots.q_i.q_{i+1}$ in such a pattern, let T_i be the type of q_i:

- if q_{i+1} is a type qualifier T_{i+1} then T_i contains at least one attribute of type T_{i+1};

– if q_{i+1} is an attribute qualifier $a_{i+1}(T_{i+1})$ then T_i contains an attribute a_{i+1} of a super-type T'_{i+1} of T_{i+1}.

As already alluded to, we allow attribute qualifiers to be declared without type, e.g., $T.\overline{q}.a$ instead of $T.\overline{q}.a(T')$. In this case we adopt a's type according to its pattern prefix $T.\overline{q}$.

3.6 T-Rule Resolution

The three ideas on precedence (nesting level, instances over types, subtyping level) can conflict. For example a deeper nesting level of a type qualifier vs. an attribute qualifier.

We use the following priorities for competing t-rules. While P1-P3 summarize the precedences we introduced in Sections 3.3-3.5, P4-P6 specify how to resolve conflicts between these.

P1 Nesting level: A natural choice consists in considering nesting level as prioritizing measure among patterns (see Section 3.3). Deeper nesting levels translate to more detailed knowledge about data-structures and thus to more specific behavior. Thus a t-rule with the pattern $T_0.T_1$ will be chosen over one with pattern T_1 for attributes of type T_1 at a path rooted at a type T_0.

P2 Instances over types: Another intuitive choice (see Section 3.4) consists in giving attribute qualifiers priority over type qualifiers. Thus $T_0.a_1(T_1)$ is chosen over $T_0.T_1$ for attribute a_1 in an event of type T_0.

P3 Subtyping level: A third intuitive choice (see Section 3.5) is to consider for any otherwise equivalent patterns the ones which use qualifiers with the most derived types: with T'_1 a strict subtype of T_1, $T_0.a_1(T'_1)$ is chosen over $T_0.a_1(T_1)$.

P4 Subtyping order: In languages like C++, C#, or Java, in which many federated distributed systems are developed, a type can have *multiple* super-types. This leads to tie-breaking issues similar to those encountered for multiple inheritance (e.g., selection from multiple methods with equivalent signatures). For instance, there might be two patterns (a) $T_0.T'_1$ and (b) $T_0.T''_1$ which both apply to a path $\langle T_0 \cdot T_1, a_1 \rangle$ where T_1 is a subtype of both T'_1 and T''_1. Assuming that the subtyping level of T_1 is the same with respect to T'_1 and T''_1 (otherwise P3) takes over, we consider the *order* of subtyping declarations for breaking ties. For instance, with T_1 **extends** $T'_1, T''_1 \ldots$ (a) will be chosen.

P5 Instances over nesting level: We need to break ties between P1 and P2). Assume that we are transforming at a path $\langle T_0 \cdot \ldots \cdot T_3, a_1 \cdot a_2 \cdot a_3 \rangle$. Now consider two matching patterns (a) $T_0.a_1(T_1).T_3$ and (b) $T_0.T_1.T_2.T_3$. Clearly, (a) is more specific than (b) according to P2, but (b) has a deeper nesting level than (a) which thus far prevails according to P1. Even if there are no attributes of type T_3 immediately in a_1, T_3 is expressed for that prefix $T_0.a_1(T_1)$ which is more specific than the corresponding prefix $T_0.T_1$ in pattern (b), and thus (a) is prioritized.

P6 Subtypes over instances: We also need to break ties between P3 and P1, and between P3 and P2. Both ties are broken by favoring subtyping over instances

(P3 over P5): we consider first a qualifier's type and then only whether it refers to an attribute or a type. Thus when comparing a type qualifier T with an attribute qualifier $a(T')$ we prioritize the first if T is a strict subtype of T'; inversely the latter. If $T = T'$ we follow P2, i.e., favor the latter.

A formal characterization of t-rule resolution semantics and arguments for its type safety are the subject of [17].

4 Contexts

This section defines contexts and higher-level abstractions for transformations. Contexts allow for grouping a common set of t-rules and types together providing better abstraction and adaptability.

4.1 Local Types and Type Mappings

Components in the targeted applications execute in given contexts. Formally, such a context is defined as follows:

Definition 5. *A* context *is a 3-tuple* $(\overline{t}, \overline{u}, \overline{r})$ *where*

- $\overline{t}$ *is a set of type definitions (see Definition 1),*
- $\overline{u}$ *is a set of type mappings of form* $T \Rrightarrow^h T'$*, where* $h \in \{?, !\}$ *indicate transformations of incoming/outgoing objects respectively*
- $\overline{r}$ *is a set of t-rules (see Definition 3).*

The previous examples did not specify any signatures for the functions referenced by t-rules. In instantiations of our model in object-oriented programming languages, methods are used as such functions. In a language with overloading/overriding, a (partial) signature can be used to distinguish between different possibilities. In general, these functions need not return values of the same type as their formal arguments, which allows for type conversions. Such conversions are desired when different interacting components use different sets of types which can not be linked via subtyping due to practical restrictions on type systems [12]. To capture this, a context c includes local type definitions. Such a definition t introduces a new type T for c.

If the type of any attribute of a transformed event object changes, then the type of the entire event has to change. A context thus includes a set of high-level *type mappings* u, each of the form $T \Rrightarrow^h T'$ indicating that instances of event type T are mapped to event type T'. h denotes whether the transformation takes place upon received (?) or sent (!) event objects. Type mappings make the programmer's intent explicit and are used for type checking t-rules as described in the next section. The type mapping $\mathsf{InvoiceLine} \Rrightarrow^! \mathsf{USInvoiceLine}$ in Figure 2 for instance represents the mapping for the logistics provider to a US context. Contexts thus decouple *which* types have to be mapped to each other (mappings) from *how* this happens (t-rules), increasing flexibility.

4.2 Context Specialization

Contexts are arranged in a hierarchy following a notion of specialization which is similar to that of inheritance in common class-based programming languages. We elaborate on the meaning of inheritance and its relation to transformation of event objects in the following:

Inheritance. Inheritance of local types, mappings, and t-rules between a parent context such as c_0 in Figure 3 and its child context c_1 obey the following rules:

- *Local types* are straighforwardly inherited by c_1.
- *Type mappings* are inherited, but can be overridden by c_1. That is, if c_0 defines $T \Rrightarrow^h T'$ for some h, T, and T', then c_1 can redefine a mapping $T \Rrightarrow^h T''$ which overrides that of c_0. Overloading can also happen, even within a same context, through the subtype-sensitivity of mappings – for example, for two types T, T' such that T' is a subtype of T, a mapping $T' \Rrightarrow^? \ldots$ takes precedence over $T \Rrightarrow^? \ldots$ for instances of T'.
- *t-rules* are similarly inherited, unless overridden. Overriding here occurs when two t-rules have *identical* patterns, otherwise the patterns co-exist (with one taking precedence over the other based on the priorities listed in Section 3.6).

The *root context* c_0 in Figure 3 is considered to represent all reference types; put differently, all types of the root context are inherited by all other contexts. In a multi-language setup, this root context would typically include types defined in an independent declaration language, with one child context per supported programming language.

Transformation. Despite a possible "chain" of context specializations, transformations in our current model are always to and from the root context as shown in Figure 3. That is, events produced in a context (e.g., c_3) are *directly* transformed to the root context c_0, if needed in any different context; events in the root context are transformed *directly* to any other context (except the original one, e.g., c_3).

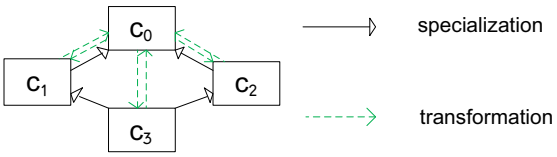


Fig. 3. Context specialization and transformation paths. Solid lines represent specialization. Dashed lines represent (optional) transformations.

While in some cases the transformation could take place indirectly by following the inheritance relation (e.g., from c_0 to c_1 to c_3 in Figure 3), this would in general raise many issues. For example, different paths could be possible (e.g., from c_0 to c_3 via c_1 or c_2), and one could expect that they produce consistent outcomes which is hard to assert.

Contexts may not always define mappings from all types T in the root context c_0 to the specific context (e.g., c_3), or from all types T' in the context c_3 to the root context. We however refrain from forcing the developer of such a context c_3 to define these mappings and transformations. After all, there may be no semantically sensible transformation. The absence of a mapping will be however noticed when compiling context c_3 and signaled as an *observation* message. We elaborate further on conformance as well as *warning* and *error* messages when discussing implementation issues shortly in Section 5.

4.3 Declaring Contexts

To remain independent of a specific programming language, we support context declarations in the widely-adopted XML (other languages are possible). Thus, developers do not need to learn new specification languages. To illustrate this, we give a brief intuition how a context for Java can be declared in XML:

```
<types>
    com. logistics .us. USInvoiceLine
    com. logistics .us. USItemSpecification
</types>
<mappings>
    <mapping   from="com.logistics.eu. InvoiceLine"
               to="com.logistics.us.USInvoiceLine"
               dir="!" />
</mappings>
<rules>
    <rule pattern="ItemSpecification"  function="toUSItemSpecification" />
    <rule pattern="InvoiceLine.Money"  function="toDollars" />
    <rule pattern="InvoiceLine. price"  function="toIdentity" />
</rules>
```

As the listing shows, all three elements of the context-tuple can easily be encoded in XML.

4.4 Practical Extensions

We provide several syntactic shortcuts for conveniently dealing with t-rules in our model. Noteworthy here are

- t-rules can be explicitly disabled upon inheritance in a child context c'. To that end, the developer can simply repeat the corresponding pattern, and use '-' in lieu of a function name. We are currently investigating labeling schemes so patterns do not need to be repeated.

– As showcased in Section 3.3, the toldentity function, it is quite convenient to exclude certain attributes from a default transformation. In many cases, this can be simpler to do than enumerating transformations individually for all non-exempt attributes. For convenience we thus provide a polymorphic toldentity function which simply returns its argument. An instance of this function in a given t-rule adopts its argument type as return type.

5 Implementation

This section presents our implementation of t-rules and contexts. We give details on rule resolution, error handling in case of invalid rules and how we improve on efficiency by generating static code.

5.1 ACTrESS

ACTrESS ("Automatic Context Transformation for Event-based Software Systems") implements our approach for the Java programming language[2]. ACTrESS is built on top of ActiveMQ [37], a fast, reliable JMS [8] broker. Our approach is implemented as a *plugin*. It intercepts event notifications passing through the broker and transforms them according to a t-rule set. Functions f used by transformations are methods invoked on notifications or their attributes, or **static** methods.

5.2 Rule Resolution and Validation

Our implementation uses a type system [17] to resolve relevant t-rules in a given context with respect to produced and consumed event types. That is, the type system performs the following tasks:

A. *Type-checking of individual t-rules.* For any t-rule $p \triangleright f$ in a given context, the type system first validates the pattern p (see Section 3.5), and then ensures that the formal argument of the function f indeed is a super-type of the expected type based on p (e.g., the expected type for a pattern $T_0.a_1(T_1)$ is T_1).
B. *Resolving t-rules.* Our type system identifies for any given event type T mapped in or out by a process all transformations for all *reachable* paths rooted at T, and retains these. This retained information is of the form $\langle T_0 \cdot \ldots \cdot T_n, a_1 \cdot \ldots \cdot a_n, f \rangle$, prompting the evaluation semantics to apply function f at the path $\langle T_0 \cdot \ldots \cdot T_n, a_1 \cdot \ldots \cdot a_n \rangle$ in any event of type T_0. These t-rules are resolved by starting from all subscribed and published types T_0, and exploring their attribute spaces recursively by following breadth (e.g., $\forall a_1$ s.t. a_1 is declared by T_0) and depth (e.g., $\forall\ a_2$ s.t. $a_1's$ type T_1 declares an attribute a_2). To deal with subtyping, for a given path (e.g., $\langle T_0 \cdot \ldots T_i, a_1 \cdot \ldots \cdot a_i \rangle$)

[2] More on ACTrESS can be found at
`http://www.dvs.tu-darmstadt.de/research/events/actress/`

the *subtype space* is explored similarly in a recursive manner (e.g., $\forall T_i' \preceq T_i$). A reachable path implies that there is no transformation for any of its prefixes; exploration does not proceed further when a transformation is resolved, as the respective function is responsible for dealing with nested attributes.

C. *Type verification.* For any t-rule involving a function f to be applied at a given path, we verify whether the type returned by f abides to the type stipulated by the mapping for the corresponding event type. Remember that there is not necessarily a 1:1 relationship between mappings and t-rules; in fact that would be undesirable in terms of expressiveness. A mapping can involve the application of multiple t-rules, and inversely, a t-rule may be applied by different mappings. Thus we do not mandate that every t-rule in a context respects all mappings for event types with paths matching the t-rule's pattern. This allows for default t-rules which typically include type qualifiers in their patterns to be overridden by more attribute-specific t-rules. The latter ones produce the correct type at a given path but the former ones – if applied there instead – would not necessarily do so.

This validation and resolution is performed at compilation, and extended at need at runtime, i.e., upon encountering new (sub)types. (Cyclic) recursions in types are handled in a way similar to iso-recursive types by *unfolding* upon *resolution* and *folding* upon *application*. That is, we halt exploration upon encountering recursion, and at a given path apply t-rules identified for prefixes of the path without the recursion. This implies that we do not support "recursion-sensitive" patterns such as $T_0.T_1.T_1$. We believe this would unnatural for programmers.

5.3 Errors and Safety

When t-rule resolution and compilation discovers invalid patterns (see Section 3.5), or failed type checks (see A. and C. above), it quits with corresponding error messages. Since resolution and compilation is done on a per-context basis, only the affected context will be unavailable (or remain unchanged if it already existed). The system will continue operating with all other contexts. We believe this is a better approach than permitting faulty t-rules and hoping that they do not trigger an runtime, or simply ignoring corresponding errors. An error in t-rule resolution is usually symptomatic of more profound inconsistencies.

Warnings are issued when a context contains several mappings with identical source (mapped) type or several t-rules with identical patterns; the last such mapping or t-rule is chosen respectively. Note that through the addition/discovery of a new subtype T' of an (attribute or event) type T no errors can be introduced, as the existing mappings and t-rules remain valid. That is, mappings remain trivially the same.There can not have been any "dormant" mapping specifically referring to T', otherwise the resolution process would have known that type (hence it's not new) and would have considered all applying t-rules. Similarly, a now active but previously disregarded t-rule must have referred to T' in its pattern already, leading again to a contradiction.

5.4 Code Generation

Besides avoiding repetitive t-rule resolution, our compilation approach has the advantage of being able to generate static code for performing transformations rather than using reflection mechanisms to dynamically invoke such functions/methods. That is, our prototype generates a class containing transformation code after analyzing t-rules and notification types. This is also the reason why we proactively explore all subtypes and retain corresponding transformations; it avoids performing any kind of resolution at runtime. We will illustrate the efficiency benefits of our approach shortly in Section 6.

5.5 Annotations

To allow for intuitive and in-code declaration, our Java prototype supports various Java annotations. To define type mappings, the developer can use the `@MapsTo` annotation, supplying the class name of the mapped class. To specify that a class should be transformed with a specific function, developers may use `@TransformWith`.

These annotations are just another form of expressing t-rules. To simplify things for developers further, we allow annotating a class's attributes with units (e.g., `@Unit("USD")`). By analyzing the units given by developers and those that the notification service uses, our prototype is able to generate the appropriate t-rules. Thus, developers can simply express their *wish* towards the data.

Our supplied annotations are not as expressive as the full t-rules but cover many typical application scenarios. Developers can use annotations to quickly generate the majority of t-rules and then fine-tune the rule set.

6 Evaluation

We evaluated our approach and implementation with regards to performance and code quality. Our results illustrate that our approach is suited for pub/sub systems by providing efficiency and extensibility while being expressive.

6.1 Performance

First we substantiate the claim made earlier that native support for transformations is beneficial for performance, by showing that (a) it is much faster than an analogous library implementation based on reflection permitting equal expressiveness and extensibility, (b) it is much faster than Apache Camel, a popular general-purpose Enterprise Application Integration (EAI) [22] framework, (b) it is as effective as manually coded transformations in application components, and (d) the application of transformations closer to producers, enabled by our model, further improves performance. Furthermore, we show that the implementation of our model scales with the number of types and t-rules in the system.

We ran the ACTrESS broker, the workload generators and data collectors in a distributed environment. We compared five setups: in *content-based*, brokers just access event content for content-based routing but do not perform any transformations, assuming all parties agree upfront on types; *model-tx* transforms event objects according to our approach, while *reflect* uses Java reflection for transformation resolution and application. Setup *camel* uses Apache Camel for transformations to investigate the impact of using EAI frameworks. EAI frameworks share the idea of integration with our approach, but do not provide implementation details. (see Section 7 for details). We use compiled transformation classes for this approach, like those that our approach generates. Finally, in the *baseline* case brokers simply forward event objects without accessing their content, illustrating the smallest latency possible.

The SPECjms2007 standard benchmark specifies a workload where distribution centers, headquarters, and suppliers form a complex supply chain and interact via various inter-company event notifications [35]. We designed a workload following SPECjms2007. Event types are taken directly from the specification. Transformations include addresses, distances, and currency.

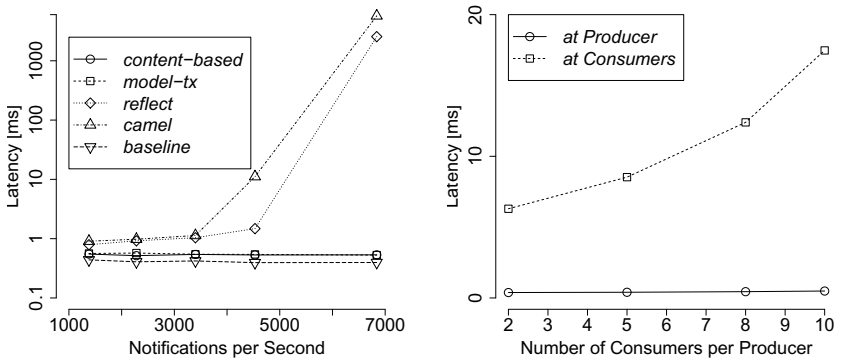
Figure 4a shows the latency for different notification rates. It is important to note that the ordinate has a logarithmic scale. As the figure shows, *reflect* adds significant latency overhead compared to our approach. This is an indicator for the increased computational effort of the reflection-based approach. Due to the increased effort, ActiveMQ cannot cope with higher event production rates; events are stalled, accounting for the steep latency increase. *Camel* performs even worse, because it is not integrated into the broker and thus additional notification marshalling and unmarshalling has to occur. This shows that even by requiring manual implementation of transformations, EAI frameworks also suffer from poor efficiency.

Scenarios *none* and *compile* achieve more than double the throughput than *reflect* and have no measurable difference, while *base* achieves even slightly more events per second. Because there is no measurable difference between *none* and *compile* and our setup already resembles the worst case where every event has to be transformed, we do not provide more details like the effect of the number of contexts.

Figure 4b illustrates the performance benefit of being able to transform closer to the producer, compared to doing it at each individual consumer. With a growing number of consumers per producer, the advantage grows. Thus, it is beneficial to be able to transform close to the producer, which our model enables.

6.2 T-rule Resolution and Code Generation Overhead

Our implementation generates transformation code from the set of rules and type mappings. While we believe that compared to the actual event notification rates, changes to the rule set, the type set or the mappings are rare, we are still interested in the overhead of this step. It is important that this step has acceptable overhead so that deployment and testing can be done quickly and – even more importantly – changes at runtime take as little time as possible.



(a) SPECjms2007-based (b) Locality

Fig. 4. Performance benefits

Figure 5 shows the time it takes to generate 1000 transformation classes for different sizes of the t-rule set and different selectivity of the rules. Selectivity means how many attributes of the type are actually affected by the rule. We used 1000 iterations to keep the variance low. Our implementation scales linearly with the number of t-rules and affected attributes. Every t-rule has to be checked because there might always be a more specific one at the end, and thus this is the optimal result. Similarly, for each affected attribute, we have to generate some code. Thus, a generator must have at least linear complexity.

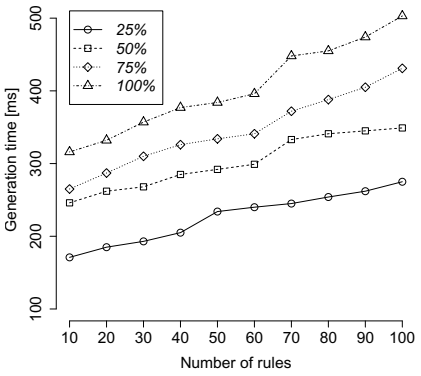


Fig. 5. Generation time

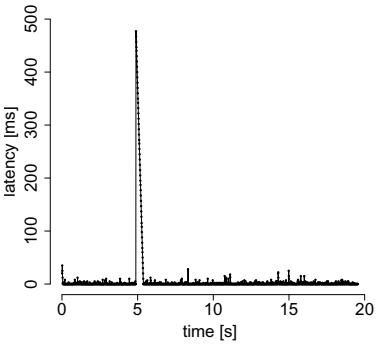


Fig. 6. Recompilation overhead

6.3 Ease of Use

Next, we demonstrate efficiency for the programmer by showing that our contexts lead to a considerably lower implementation effort than manual coding of transformations, and that existing transformations can be changed more easily. We use five typical event-based applications to compare the required lines of code and ease of changes.

Kemerer [23] analyzed over 30 software complexity metrics and concluded, that "a number of the more complex metrics may be essentially measuring the size of the program or other component under investigation, and therefore may provide little additional information". We thus see lines of code as a valid indicator for code complexity and maintenance effort and used it for our comparison.

To give a brief comparison between the two approaches, consider the example to transform attributes of type Address. Our approach needs just one rule to transform every occurrence of an attribute of type Address. When coding transformations manually, the developer has to write a dedicated **if-else** branch for every event type with an address attribute (directly or indirectly). Inside, several lines of code for attribute extraction, object creation and transformation are needed.

For our comparison, we use the event types specified by various applications: The SPECjms2007 benchmark introduced in Section 6.1. Transformations include address translations between different regional formats and changing product descriptions (in orders, invoices, etc.) to conform to each site's expectations.

The MARKETCETERA[3] automated trading platform defines various event types capturing stock ticker quotes and allows for elaborate automated trading. The transformations on this platform perform currency conversions, timestamp formatting changes and renaming of some indicators, assuming differing terminology on the consumers.

The HTM traffic management system handles data from sensors and cameras along streets and highways, monitoring road, traffic and weather conditions [34]. Such systems are used by many large cities. Transformations include coordinate translation, timestamp format changes and unit conversion.

DRADEL is an application environment for modeling and analyzing distributed architectures, including code generation [25]. Since it runs on top of a message-oriented middleware, multi-user support is possible. In such a setting however, path references and line numbers need to be adapted to each platform. Thus, operational events of DRADEL need to be transformed accordingly.

The Emergency Response System (ERS) is a distributed application running on multiple mobile devices and helps organizing human resources during natural disasters [31]. Its events often refer to geographical regions, which need transformation between individual users to adapt to their specific format.

Table 1 compares the lines of code needed to *specify* transformations by manual coding and by our approach, showing a clear benefit for the latter. The numbers indicate the effort necessary to specify the transformations of one client that needs to transform events. We did not count lines of code that can be generated

[3] http://www.marketcetera.com

Table 1. Code complexity comparison

	manual			*ACTrESS*		
	specify	*change*	*extend*	*specify*	*change*	*extend*
SPECjms2007	167	15	92	22	2	1
marketcetera	108	9	37	16	2	1
HTM	237	16	133	21	2	1
DRADEL	417	16	139	30	3	2
ERS	351	12	117	21	3	2

by standard IDEs. For additional clients needing different transformations, the effort has to be made again, multiplying our benefits. Furthermore, the *change* columns show the number of *individual places* in the code that needed to be changed when a certain type (e.g., Address) should be changed into a different format.

Contrary to intuition, EAI frameworks like Apache Camel do not reduce complexity at this step. Although supporting event transformations architecturally, transformations still need to be coded manually, resulting in above depicted effort.

6.4 Extensibility

Changes can roughly occur in two ways: transformation functions need to be changed or new types are introduced. In case of changed transformation functions, one can simply adapt them (e.g., change toUSAddress). Adding new types is more complicated. Suppose we want to add a sensor to the traffic management system to detect oil on the road (raising an Oil event). Every sensor event in the system has an attribute SensorMetadata which has an attribute of type Location. In the manual approach, it is thus not immediately apparent that adding the Oil event requires a new piece of transformation code. A developer will have to analyze the existing transformation code and realize that locations are transformed and then write the new code. With our approach, only the mapping has to be defined. The *extend* column in Table 1 illustrates this by giving the required number of lines of code for the necessary analysis.

In case of changes at runtime, ACTrESS dynamically recompiles the generated classes when changes occur. Figure 6 illustrates the impact on performance. It shows that there is a brief increase in latency for the recompilation, after which the system performs as before. This demonstrates that our implementation can handle changes at runtime without significant impact on performance.

7 Related Work

In this section, we divide the space of object transformations along different dimensions and relate existing work to our proposed approach along these dimensions.

7.1 Transformation Space

As mentioned there are different dimensions along which one can divide the space of object transformations. Among these, without attempting to be exhaustive but to cover the main related work, we can consider *granularity*, *completeness* and *topology*. The possibilities for *granularity* for instance include

G1 Monolithic object transformations. Objects of given event types are transformed as a whole.
G2 Attribute-wise transformations. Event objects are transformed attribute-wise, with a 1-1 mapping of attributes.
G3 Nested attribute-wise transformation. With objects containing attributes that are objects, one can allow attribute-wise transformation of such nested objects.
G4 Path-based transformation. Transformations can be expressed on any attributes, at any nesting level, in objects.

In terms of the actual transformation, there are also different levels of computational *completeness* that one can imagine:

C1 Type or meta-data transformation. Objects retain their actual state, but they are converted to other types. This includes also traditional subtype subsumption, where simply a subset of the attributes are retained when accessing an object via a super-type.
C2 Lookup-based transformation. An object is substituted by another one based on a lookup in a static or dynamic data-structure. Such objects correspond to discrete values.
C3 Function-based transformation. A function is invoked with an object and can perform any computations to construct a substitute object.
C4 Function-based stateful transformation. Same as above, except that the function can also persist state in variables.

There are also different places in the *topology* of a distributed application for transformation application, e.g.,

T1 Peer-based transformation. Every application component or process performs its own transformations on incoming — maybe also outgoing — event objects.
T2 Distributed transformation. A distributed middleware system performs transformations on conveyed event objects, through a dedicated server or component.
T3 Decentralized transformation. Here a distributed middleware performs transformations without relying on a centralized component.

Our solution presented supports the highest level for any of these criteria, i.e., G4, C4, and T3.

7.2 Existing Work

In database integration, data from one database is transformed to adhere to the schema of another database [9,30] (T2). Anonymity in a federated pub/sub-based system prevents *schema integration* used by these approaches. A subscriber does not know who produced an event it receives and thus not the schema it follows.

Chung [14] argues that it is infeasible to decide upon a database for a whole organization, proposing DATAPLEX as a middleware layer implemented as centralized data mediation component (T2) to allow uniform access to databases.

Cluet et al. [16] address the issue of integrating heterogeneous data sources by proposing a rule language for conversion between various data representations. The system is designed for request/reply communication while we focus on data distributed via publish/subscribe, where subscribers may not know the origin (communication endpoint) of data. Cilia et al. [15] propose a solution to deal with heterogeneous data sources in pub/sub systems using a self-describing model. Neither of the above however verifies typing of transformations. Dozer [3] supports mapping of data objects between Java Beans. Dozer supports expressive and complex mappings; conversion resolution and execution occur via Java Reflection at runtime, limiting performance and safety.

Foster et al. [18] present a bi-directional tree transformation approach. Their transformation functions allow to mediate between different views of same data where updates are applied backwards to the original data (C3). In contrast, our approach purposely supports uni-directional and non-deterministic transformations. Every subscriber gets its own copy of an event and thus events are not meant to be shared like documents.

Several authors propose structural subtyping to decouple components (G3, C1), which has been promoted by several research programming languages (e.g., Lingua Franca [27], Accute [36]). Whiteoak [21] extends Java with structural conformance similarly to *compound types* [12].

HydroJ [24] extends Java with relaxed conformance on nested semi-structured events exchanged between processes. Similarly, most publish/subscribe systems follow the model described in [29] which promotes the use of hash maps to convey events in the form of $\langle key, value \rangle$ pairs. This places all burden on programmers as these need to manually inspect, marshal/unmarshal and transform event objects at generation and reception. With Hashtypes [36] type representations are hashed, including "contents" of corresponding instances, function signatures in modules, etc. Hashes are propagated with objects. Given the focus on point-to-point and not implicit communication, transformations are applied at end components (T1). None of these approaches support value-based transformations (C1).

Platforms like Sun RPC [38], OMG's CORBA [28], or Web Services [11] only mediate between *encodings* of values (e.g., little vs. big endian).

Java Internationalization (JI) [4] provides support for context-specific interpretation of precise data types (e.g., strings in different character sets, times in different zones). JI also supports automatic translation of character strings between different natural languages based on dictionaries (C2). JI furthermore

includes a framework which allows application-specific types (*resource bundles*) to be interpreted differently across contexts (*locales*). In combination with Java Remote Method Invocations (RMI) [5] or other remote communication paradigms, JI can hence be used as a foundation to address similar problems as studied herein. However, the JI framework consists merely in an API, while the present work aims at providing intuitive and safe mechanisms for *implementing* such an API. .NET Internationalization [6] provides analogous functionalities to JI for the .NET platform. Similarly, design and architectural patterns such as *adapters* [26] provide a locus between application components to perform transformations but do not provide support for actually implementing them.

Enterprise Application Integration (EAI) specifies *message transformations* [22] to deal with heterogeneity. However, EAI just specifies a pattern (in fact, they can be seen as a more detailed definition of adapters), without any suggestions regarding its implementation. EAI Frameworks like Apache Camel [2] thus support message transformations, but provide merely an API with little support for implementing it. Transformations take place on the entire message bodies (`G1`) and there is no facility for a *canonical data model*, leaving its design to the programmer. As demonstrated, efficiency is poor. Microsoft BizTalk [33] provides support for transformation with XSLT and *orchestration*s. However, mappings are static, take place on entire message bodies (`G1`) and new producers or consumers have to be added explicitly. Other EAI frameworks expose similar limitations.

8 Conclusions

We have introduced a foundational model for interoperability in federated distributed software based on transformations. Our model is expressive in that it supports the whole spectrum of transformations including enrichment of events and allows other type conformance models to be implemented atop. As we have demonstrated it is easy to use by supporting fine-grained expression of transformations as opposed to monolithic ones, and it allows for extensibility at runtime, while at the same time showing being efficient, causing no measurable overhead on an underlying content-based pub/sub system. Last but not least, our approach is safe, by promoting clear semantics for transformations, whose application is verified and determined first and enforced at runtime.

We are currently working on supporting clients in other languages, as well as on an implementation of type versioning on top of our model. We are also investigating extensions to our model including nested transformations. These will allow functions used for transformations to explicitly re-invoke the transformation process in order to avoid invoking or repeating transformation functions for nested attributes. Care must be taken here to not increase expressiveness at the cost of simplicity. Last but not least, we are working on optimal placement of transformation operations in decentralized publish/subscribe networks.

References

1. Amazon Simple Notification Service, http://aws.amazon.com/sns/
2. Apache Camel Type Convert, http://camel.apache.org/type-converter.html
3. Dozer, http://dozer.sourceforge.net
4. Java Internationaliation, http://java.sun.com/javase/technologies/core/basic/intl/
5. Java Remote Method Invocation, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html
6. .NET Internationalization, http://msdn.microsoft.com/en-us/goglobal/bb688096.aspx
7. Project Kafka, http://engineering.linkedin.com/tags/kafka
8. Java Message Service - Specification, version 1.1. Tech. rep., Oracle Co. (2008), http://www.oracle.com/technetwork/java/jms/index.html
9. Adair, J., Coyle Jr, D., Grafe, R., Lindsay, B., Reinsch, R., Resch, R., Selinger, P., Zimowski, M.: Heterogenous Database Communication System in Which Communicating Systems Identify Themselves and Convert any Requests/Responses Into Their own Data Format. US Patent Number 5,416,917 (1995)
10. Allen, E., Hilburn, J., Kilpatrick, S., Luchangco, V., Ryu, S., Chase, D., Steele, G.: Type Checking Modular Multiple Dispatch With Parametric Polymorphism and Multiple Inheritance. In: OOPSLA 2011 (2011)
11. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. Springer (2003)
12. Büchi, M., Weck, W.: Compound Types for Java. In: OOPSLA 1998 (1998)
13. Buchmann, A., Pfohl, H.C., Appel, S., Freudenreich, T., Frischbier, S., Petrov, I., Zuber, C.: Event-Driven Services: Integrating Production, Logistics and Transportation. In: SOC-LOG 2010 (2010)
14. Chung, C.W.: DATAPLEX: An Access to Heterogeneous Distributed Databases. CACM 33, 70–80 (1990)
15. Cilia, M., Antollini, M., Bornhövd, C., Buchmann, A.: Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach. In: DEBS 2004 (2004)
16. Cluet, S., Delobel, C., Siméon, J., Smaga, K.: Your Mediators Need Data Conversion! In: SIGMOD 1998 (1998)
17. Eugster, P., Freudenreich, T., Frischbier, S., Appel, S., Buchmann, A.: Sound Transformations for Message Passing Systems. Tech. rep. (2012), http://www.dvs.tu-darmstadt.de/publications/pdf/transsem.pdf
18. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem. In: POPL 2005 (2005)
19. Freudenreich, T., Frischbier, S., Appel, S., Buchmann, A.: ACTrESS - Automatic Context Transformation in Event-Based Software Systems. In: DEBS 2012 (2012)
20. Frischbier, S., Michael, G., Mayer, D., Roth, A., Webel, C.: Emergence as Competitive Advantage - Engineering Tomorrowś Enterprise Software Systems. In: ICEIS 2012 (2012)
21. Gil, J., Maman, I.: Whiteoak: Introducing Structural Typing Into Java. In: OOPSLA 2008 (2008)
22. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (2004)
23. Kemerer, C.: Software Complexity and Software Maintenance: A Survey of Empirical Research. Annals of Software Engineering 1, 1–22 (1995)
24. Lee, K., LaMarca, A., Chambers, C.: HydroJ: Object-oriented Pattern Matching for Evolvable Distributed Systems. In: OOPSLA 2003 (2003)

25. Medvidovic, N., Dashofy, E., Taylor, R.: The Role of Middleware in Architecture-based Software Development. International Journal of Software Engineering and Knowledge Engineering 13(4), 367–393 (2003)
26. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a Taxonomy of Software Connectors. In: ICSE 2000 (2000)
27. Muckelbauer, P.A., Russo, V.F.: Lingua Franca: An IDL for Structural Subtyping Distributed Object Systems. In: COOTS 1995 (1995)
28. Object Management Group: CORBA, http://www.corba.org
29. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus - An Architecture for Extensible Distributed Systems. In: SOSP 1993 (1993)
30. Parent, C., Spaccapietra, S.: Issues and Approaches of Database Integration. Commun. ACM 41, 166–178 (1998)
31. Popescu, D., Garcia, J., Bierhoff, K., Medvidovic, N.: Impact Analysis for Distributed Event-based Systems. In: DEBS 2012 (2012)
32. Reiss, S.P.: Connecting Tools Using Message Passing in the Field Environment. IEEE Software 7(4), 57–66 (1990)
33. Rosanova, D.: Microsoft BizTalk Server 2010 Patterns. Packt. Pub. Limited (2011)
34. Schneider, S.: DDS and the Future of Complex, Distributed Data-Centric Embedded Systems (2006), http://www.eetimes.com/design/embedded/4025967/DDS-and-the-future-of-complex-distributed-data-centric-embedded-systems
35. Sachs, K., Kounev, S., Bacon, J., Buchmann, A.: Performance Evaluation of Message-oriented Middleware using the SPECjms2007 Benchmark. Performance Evaluation 66(8), 410–434 (2009)
36. Sewell, P., Leifer, J.J., Wansbrough, K., Nardelli, F.Z., Allen-Williams, M., Habouzit, P., Vafeiadis, V.: Acute: High-level Programming Language Design for Distributed Computation. J. Funct. Program. 17(4-5), 547–612 (2007)
37. Snyder, B., Bosanac, D., Davies, R.: ActiveMQ in Action. Manning Publications Co. (2011)
38. Srinivasan, R.: RPC: Remote Procedure Call Protocol Specification Version 2 (1995)
39. Sullivan, K., Notkin, D.: Reconciling Environment Integration and Software Evolution. ACM Trans. Softw. Eng. Methodol. 1(3), 229–268 (1992)

RedCard: Redundant Check Elimination for Dynamic Race Detectors

Cormac Flanagan[1] and Stephen N. Freund[2]

[1] University of California at Santa Cruz
[2] Williams College

Abstract. Precise dynamic race detectors report an error if and only if an observed program trace exhibits a data race. They must typically check for races on all memory accesses to ensure that they catch all races and generate no spurious warnings. However, a race check for a particular memory access is guaranteed to be *redundant* if the accessing thread has already accessed that location within the same *release-free span*. A release-free span is any sequence of instructions containing no lock releases or other "release-like" synchronization operations, such as `wait` or `fork`.

We present a static analysis to identify redundant race checks by reasoning about memory accesses within release-free spans. In contrast to prior whole program analyses for identifying accesses that are always race-free, our redundant check analysis is span-local and can also be made method-local without any major loss in effectiveness. RedCard, our prototype implementation for the Java language, enables dynamic race detectors to reduce the number of run-time checks by close to 40% with no loss in precision.

We also present a complementary *shadow proxy* analysis for identifying when multiple memory locations can be treated as a single location by a dynamic race detector, again with no loss in precision. Combined, our analyses reduce the number of memory accesses requiring checks by roughly 50%.

1 Introduction

Multithreaded programs are prone to race conditions caused by unintended interference between threads, a problem exacerbated by the broad adoption of multi-core processors. A race condition occurs when two threads concurrently perform *conflicting* memory accesses that read or write the same location, where at least one access is a write. The order in which the conflicting accesses are performed may affect the program's subsequent state and behavior, likely with unintended or erroneous consequences. Such problems may arise only on rare interleavings, making them difficult to detect, reproduce, and eliminate.

The problems caused by data races have motivated much work on detecting races via static [1, 3, 5, 15, 23, 30, 40] or dynamic [10, 14, 31, 32, 35, 37, 43] analysis, as well as via post-mortem analyses [2, 9, 34]. In this paper, we focus on

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 255–280, 2013.

on-line dynamic race detectors, which detect races by monitoring a program as it executes. Dynamic race detectors typically use a broad notion of when two conflicting accesses are considered concurrent to maximize coverage, and conflicting accesses need not be performed at exactly the same time. Instead, a race condition is said to occur when there is no "synchronization dependence" between the two accesses, such as the dependence between a lock release by one thread and a subsequent lock acquire by a different thread. These various kinds of synchronization dependencies form a partial order over the instructions in the execution trace called the *happens-before relation* [26]. Two memory accesses are considered to be *concurrent* if they are not ordered by this relation.

Dynamic detectors may be classified by whether they are *precise* or *imprecise*. Precise analyses never produce false alarms. Instead, they compute an exact representation of the happens-before relation for the observed trace and report an error if and only if the observed trace has a race condition [18, 28, 32].[1] Despite the development of a variety of implementation techniques (including vector clocks [28, 32], epochs [18], accordion clocks [10], and others [14]), the overhead of precise dynamic race detectors can still be prohibitive.

A promising approach for improving the performance of precise dynamic race detectors is to use a static analysis to identify accesses that do not need to be checked at run time. Several prior analyses identify accesses that are guaranteed to be *race-free*, with good results [8, 17, 30, 38], but the more effective analyses are typically whole-program and quite expensive.

We focus instead on the orthogonal and more tractable property of identifying race checks that are guaranteed to be *redundant*, where a race check is redundant if ignoring that access during dynamic race checking leads to no missed races or false alarms. Interestingly, whereas verifying an access to be statically race-free typically requires information about multithreaded control flow, aliasing, and synchronization most readily computed via whole-program analysis, many race checks can be verified as redundant using more local information, specifically the locations the current thread has accessed since entering the current *release-free span*. Informally, a release-free span is a sequence of instructions containing no lock releases (or other synchronization operations such as `fork` that may similarly introduce an outgoing edge in the happens-before graph).

We present a static analysis for identifying and eliminating redundant race checks based on this notion. While others have explored removing limited forms of redundant checks in various imprecise detectors [8, 13] and in programs with structured parallelism [33], we focus exclusively on redundancy (independent of the dynamic race detection algorithm used) and tailor our analysis to be highly effective at reasoning about that notion. It primarily leverages local reasoning about memory accesses, aliasing, data invariants, and synchronization to

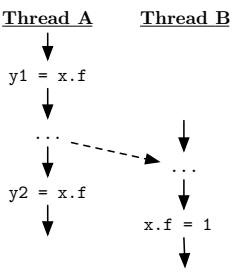
[1] Precise dynamic race detectors do not reason about all possible traces and may not identify races occurring on unobserved code paths. While full coverage is desirable, it comes at the cost of potential false alarms due to the undecidability of the halting problem. To avoid false alarms, precise race detectors thus focus on detecting race conditions only on the observed trace.

eliminate a substantial amount of redundant checking for any dynamic race detector, with no loss in precision.

Redundant Check Elimination. A *release-free span* is a code fragment containing no operations that create out-going edges in the happens-before graph. Thus release-free spans may not contain lock releases (which create happens-before edges from the release to any subsequent acquires by other threads), forks (which create edges from the fork to the first step of the new thread), writes to volatiles (which create edges from the write to subsequent reads), and so on.

Spans exhibit a key property for our analysis: if an access to a memory location is in a race with a step by another thread, all previous accesses to that memory location within the current span will also be in a race with the other thread.

To illustrate why this property holds, suppose Thread A executes some span reading `x.f` twice while Thread B writes to `x.f`. If the write is in a race with the second read, then it must also be in a race with the first. If this were not the case, then the happens-before graph for the racy execution would have the form shown on the right, where the dashed line is present due to the synchronization operation ensuring that the write happens after the first read. However, this situation is impossible, because release-free spans contain no out-going edges to steps by other threads, and so the dashed line cannot exist. Thus it is sufficient for a dynamic race detector to check for races only on the first access to a memory location in each span. It may ignore any subsequent accesses without any loss in precision.



At a high level, our analysis records at each program point a context Π containing *available paths* describing memory locations previously accessed in the current release-free span, as well as other local state invariants we describe later. To illustrate this idea, consider the code to the right in which lines 1–3 form a release-free span. Our system verifies that `x.f` is an available path at line 3, and thus labels only the first access in the span as `Check` to indicate that the dynamic detector must examine that access. No paths are available after the span ends on line 4, and the detector must again check the first access to `x.f` in the new span.

```
1 synchronized(m) {
2   y = x.f^Check;
3   y = x.f^NoCheck;
4 }
5 y = x.f^Check;
```

To support arrays, contexts may also include universally quantified paths, as in the code in Figure 1, which clears a two-dimensional array. The comments indicate the most salient context items inferred at various program points. Examining the inner loop on lines 3–6, the `Check` annotation for assignment "`a[i]`$^{\texttt{NoCheck}}$`[j]`$^{\texttt{Check}}$ `= 0`" on line 5 indicates that only a single check, on the access to the `j`th element of array `a[i]`, is required on each iteration of the loop. Race checks on the access `a[i]` on 5 are redundant, because that location was previously checked, as indicated by its presence in the context on line 4. Removing these checks substantially reduces the number of run-time checks

```
1 // Π : ∅
2 for (int i = 0; i < a.length; i++) {
3   for (int j = 0; j < a[i]^Check.length; j++) {
4     // Π : a[i], ∀(k ∈ 0 to j). a[i][k]
5     a[i]^NoCheck[j]^Check = 0;
6   }
7   // Π : ∀(h ∈ 0 to i). ∀(j ∈ 0 to a[h].length). a[h][j]
8 }
9 // Π : ∀(i ∈ 0 to a.length). ∀(j ∈ 0 to a[i].length). a[i][j]
```

Fig. 1. Redundant Check Elimination for Arrays

performed. The post-loop context on 9 shows that any subsequent accesses to this two-dimensional array within the current span would also be check-free.

Shadow Proxies. Our analysis also identifies memory locations for which no checks are ever required, which further reduces time and space overhead since no analysis (or *shadow*) state needs to be maintained for such locations. Our approach is based on the observation that accesses to different memory locations are often correlated. For example, a `Point` object may be used in such a way that whenever its `x` field is accessed, its `y` field is accessed *in the same span*. We say that `y` is a shadow proxy for `x` in this case, and any race on `x` naturally implies that there is a race on `y`. A dynamic race detector will thus still detect the presence of a race, even if all accesses to `x` are ignored. Our analysis also identifies shadow proxy relationships between array elements.

RedCard. We have implemented our analysis in RedCard (Redundant Checks for Race Detectors) for Java bytecode programs. On a collection of benchmarks, RedCard reduced the number of run-time race condition checks required by a precise detector by roughly 40%. When configured to also infer shadow proxies, RedCard reduced the number of checks by close to 50%. Eliminating these redundant checks in the FastTrack dynamic race detector [18] improved its running-time by about 25%.

A number of other tools, such as Chord [30], leverage global may-happen-in-parallel or other flow-insensitive analyses to reason about conflicting accesses. This can be quite effective at finding unnecessary checks in some programs, but typically requires more expensive and less scalable global reasoning. We compare RedCard to Chord-like analyses in more detail in our experimental validation.

Contributions. In summary, this paper:

- defines a notion of a redundant check for precise dynamic race detectors, describes an analysis to identify redundant checks as an effect system for an idealized language, and proves this analysis is correct (Sections 2 and 3);
- extends the core analysis to handle arrays (Section 4) and to identify shadow proxies, which characterize memory locations that can be ignored by race detectors entirely, with no loss in precision (Section 5);
- describes our RedCard system for inferring redundant accesses in Java programs (Section 6); and

$$
\begin{array}{rll}
\mathcal{P} \in \mathit{Program} & ::= & \overline{D}\ \ s_1 \| \ldots \| s_n \\
D \in \mathit{Defn} & ::= & \texttt{class}\ C\ \texttt{\{}\ \overline{f}\ \ \overline{\mathit{meth}}\ \texttt{\}} \\
\mathit{meth} \in \mathit{Method} & ::= & m\texttt{(}\overline{x}\texttt{)}\ \mathit{spec}\ \texttt{\{}\ s;\ \texttt{return}\ r\ \texttt{\}} \\
\\
s \in \mathit{Stmt} & ::= & \texttt{skip} \mid s; s \mid \texttt{if}\ \mathit{be}\ s\ s \mid \texttt{while}\ \mathit{be}\ s \mid x\ \texttt{=}\ e \mid x\ \texttt{=}\ \texttt{new}\ C \\
& & \mid y.f^k\ \texttt{=}\ x \mid x\ \texttt{=}\ y.f^k \mid x\ \texttt{=}\ y.m\texttt{(}\overline{z}\texttt{)} \mid \texttt{acq}\ x \mid \texttt{rel}\ x \\
\\
e \in \mathit{Expr} & ::= & x \mid v \mid \ldots \\
\mathit{be} \in \mathit{BoolExpr} & \subseteq & \mathit{Expr} \\
v \in \mathit{Value} & ::= & \rho \mid \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid \ldots \\
k \in \mathit{CheckOption} & ::= & \texttt{Check} \mid \texttt{NoCheck} \\
\\
C \in \mathit{ClassName} & & f \in \mathit{FieldName} \qquad m \in \mathit{MethodName} \\
x, y, r \in \mathit{Var} & & \rho \in \mathit{Address}
\end{array}
$$

Fig. 2. RedJava Syntax

– shows that, on a collection of benchmarks, RedCard reduces the number of access checks required by a precise detector by close to 50%, leading to a roughly 25% speedup in the FastTrack race detector (Section 7).

2 RedJava Language and Semantics

We formalize our analysis for the idealized language RedJava, a multithreaded subset of Java summarized in Figure 2. A RedJava program $\mathcal{P}$ is a sequence of class definitions $\overline{D}$ together with a sequence of statements $s_1 \| \ldots \| s_n$. At run time, the statements in $s_1 \| \ldots \| s_n$ are evaluated concurrently by multiple threads.

Each definition associates a class name C with a collection of field and method declarations. Field declarations are simply names and contain no type information. (We assume that standard typing requirements are verified for RedJava programs via a separate analysis. Enforcing a traditional typing discipline is orthogonal to the concerns of this paper and omitted for simplicity.) A method declaration "$m(\overline{x})$ *spec* `{` s; `return` r `}`" defines a method m with parameters $\overline{x}$ and statement body s. The method returns the value stored in variable r. The variable `this` is implicitly bound to the receiver in the method body. We assume that all methods have unique names to avoid type-based method resolution. Methods also contain specifications, as described below.

RedJava statements are expressed in a low-level language somewhat analogous to JVM bytecode. Statement forms include sequential composition, conditionals, while loops, and method calls. Local variables, which are not explicitly declared, can be mutated via the assignment statement "`x =` e". We leave the set of expressions e intentionally unspecified but assume that they range over at least `null`, boolean values, variable identifiers, and object addresses.

The object allocation statement "`x = new C`" assigns a freshly allocated C object to variable `x`, where all fields of that object are initialized to `null`. Field read

(`x = y.f`^{k}) and write (`y.f`^{k} `= x`) statements includes a check tag k, which is `Check` if the dynamic race detector should verify it for race-freedom and `NoCheck` if the dynamic race detector should skip verifying that access. As in Java, each object x has a corresponding mutual exclusion lock with operations `acq` x and `rel` x.

A program $\overline{D}\ s_1 \| \ldots \| s_n$ executes by evaluating the statements $s_1, \ldots, s_n$ in concurrent threads. A companion technical report [19] formalizes a small-step semantics describing evaluation as a relation $\overline{D} \vdash \Sigma \rightarrow^a \Sigma'$, where the run-time state Σ stores a heap of dynamically allocated objects; and Σ' is the same heap updated with the effects of this step. The *Action* a records any shared-memory or synchronization operation performed by the step. For example, the action $t{:}\texttt{acc}(\rho.f^{\texttt{Check}})$ denotes that the thread identifier $t \in \mathit{Tid}$ performed a checked access to the field `f` of the object at address ρ. The special action $t{:}\epsilon$ denotes a step that has no heap effect.

$$
\begin{array}{rcl}
u, t \in \mathit{Tid} & ::= & 1 \mid 2 \mid \ldots \\
a, b \in \mathit{Action} & ::= & t{:}\texttt{acc}(\rho.f^k) \mid t{:}\texttt{acq}(\rho) \mid t{:}\texttt{rel}(\rho) \mid t{:}\epsilon \\
\alpha \in \mathit{Trace} & ::= & \overline{\mathit{Action}}
\end{array}
$$

The relation $\overline{D} \vdash \Sigma \rightarrow^\alpha \Sigma'$ is the reflexive transitive closure of the single-step relation and formalizes the behavior of a *Trace* $\alpha = a_1 \cdot a_2 \cdots a_n$. The initial state Σ_0 contains a freshly-allocated object for each free variable in $s_1 \| \ldots \| s_n$. Those objects may be referenced by multiple threads.

2.1 Race Conditions and Dynamic Race Detection

The *happens-before* relation for trace α is the smallest reflexive, transitive relation $<_\alpha$ on α such that $a <_\alpha b$ if a occurs before b in α and either: a and b are performed by the same thread; or a releases some lock and b acquires that lock.

Two operations are *concurrent* if they are not ordered by the happens-before relation, and two accesses *conflict* if they read or write to the same location $\rho.f$. Our definition of conflicting accesses implies that two reads may conflict, which simplifies our formal development. We address commuting read operations in Section 6. A trace has a *race condition* if it has a pair of concurrent conflicting accesses. Moreover, a trace has a *detected race condition* if it has a pair of concurrent conflicting accesses that are both marked as `Check`.

We now consider which accesses in a trace require checks. A *release-free span*, or simply a *span*, is the sequence of instructions by a thread between two release statements. A trace is *well-formed* if each unchecked access $t : \texttt{acc}(\rho.f^{\texttt{NoCheck}})$ is preceded by a checked access $t{:}\texttt{acc}(\rho.f^{\texttt{Check}})$ in the same span, under the assumption the trace prefix up to the unchecked access is race-free.

To motivate this race-free assumption, consider the code fragment to the right. We would like to annotate the last read `y2.g` as `NoCheck`, arguing that `y1` and `y2` are aliases, and that `y1.g` was previously read. However, concurrent racy writes to `x.f` could cause `y1`

```
y1 = x.f^Check;
z1 = y1.g^Check;
y2 = x.f^NoCheck;
z2 = y2.g^NoCheck;
```

$$
\begin{array}{lll}
\Pi \in \mathit{Context} & ::= & \overline{\pi} \\
\pi \in \mathit{ContextItem} & ::= & p \mid c \mid be \\
c \in \mathit{AliasConstraint} & ::= & x = p
\end{array}
\qquad
\begin{array}{lll}
P, Q \in \mathit{PathSet} & ::= & \overline{p} \\
p \in \mathit{Path} & ::= & x.f \\
K \in \mathit{ModifiesSet} & ::= & \overline{\kappa} \\
\kappa & ::= & f \mid \textsc{All}
\end{array}
$$

Fig. 3. Contexts, Path Sets, and Modifies Sets

and `y2` to differ. The race-free prefix assumption precludes this possibility and enables the unchecked read of $\texttt{y2.g}^{\texttt{NoCheck}}$.

The following theorem, which is proved in the companion technical report, shows that the checked accesses in well-formed traces are sufficient to guarantee that, for any trace with one or more race conditions, at least one of those races will be detected. In practice, our implementation detects all races on all of our benchmarks, since the above corner case in which one race masks another race is extremely rare.

Theorem 1 (Race Detection). *Any well-formed trace has a race condition if and only if it has a detected race condition.*

3 Redundant Check Elimination

We now develop a static analysis for identifying which accesses are redundant (that is, race-free race under the assumption that previous accesses are race-free). For presentation purposes, we describe our analysis as a decision procedure for verifying `NoCheck` annotations, but this decision procedure naturally maps to an inter-procedural least-fixed point algorithm for inferring `NoCheck` annotations in our implementation, as described in Section 6.

Our analysis tracks a context Π that includes paths p of the form $x.f$ that have already been accessed in the current span.[2] In the simple case, redundant check elimination can be accomplished by standard compiler optimizations for redundant load elimination (see, *e.g.* [4, 24, 39]), which may remove the second redundant read of $x.f$ entirely. However, redundant check elimination applies in more general cases, for example when iterating through an array for a second time in a span, since the array contents likely would not fit in the register file, or on a read that occurs after different writes on different control-flow paths.

To help identify redundant checks, the context includes must-alias information, such as the equality $\texttt{y1} = \texttt{y2}$ from the earlier example in Section 2.1. Finally the context includes boolean constraints over local variables, which additionally aid in reasoning about both aliasing and array accesses, as discussed below.

To facilitate modular reasoning, each method carries a specification

$$\mathit{spec} \in \mathit{MethodSpec} = \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K$$

[2] We discuss longer access paths, such as `a.f[i]`, in Section 6.

$\boxed{\overline{D} \vdash s : \Pi \rightsquigarrow \Pi'}$

[Skip] $\overline{D} \vdash \texttt{skip} : \Pi \rightsquigarrow \Pi$

[Acq] $\overline{D} \vdash \texttt{acq}\ x : \Pi \rightsquigarrow \Pi$

[Rel] $\overline{D} \vdash \texttt{rel}\ x : \Pi \rightsquigarrow \Pi \setminus \textsc{All}$

[Assign] $\overline{D} \vdash x\ \texttt{=}\ e : \Pi \rightsquigarrow \Pi[x := w] \cup \{x = e[x := w]\}$

[New] $\overline{D} \vdash x\ \texttt{=}\ \texttt{new}\ C : \Pi \rightsquigarrow \Pi[x := w]$

[Read]
$$\frac{k = \texttt{NoCheck} \Rightarrow \Pi \vdash y.f \qquad p = (y.f)[x := w] \qquad \Pi' = \Pi[x := w] \cup \{x = p, p\}}{\overline{D} \vdash x\ \texttt{=}\ y.f^k : \Pi \rightsquigarrow \Pi'}$$

[Write]
$$\frac{k = \texttt{NoCheck} \Rightarrow \Pi \vdash y.f \qquad \Pi' = (\Pi \setminus f) \cup \{x = y.f, y.f\}}{\overline{D} \vdash y.f^k\ \texttt{=}\ x : \Pi \rightsquigarrow \Pi'}$$

[If]
$$\frac{\overline{D} \vdash s_1 : \Pi \cup \{be\} \rightsquigarrow \Pi_1 \qquad \overline{D} \vdash s_2 : \Pi \cup \{\neg be\} \rightsquigarrow \Pi_2}{\overline{D} \vdash \texttt{if}\ be\ s_1\ s_2 : \Pi \rightsquigarrow (\Pi_1 \sqcap \Pi_2)}$$

[While]
$$\frac{\Pi \sqsubseteq \Pi_{inv} \qquad \Pi' \sqsubseteq \Pi_{inv} \qquad \overline{D} \vdash s : (\Pi_{inv} \cup \{be\}) \rightsquigarrow \Pi'}{\overline{D} \vdash \texttt{while}\ be\ s : \Pi \rightsquigarrow (\Pi_{inv} \cup \{\neg be\})}$$

[Seq]
$$\frac{\overline{D} \vdash s_1 : \Pi \rightsquigarrow \Pi'' \qquad \overline{D} \vdash s_2 : \Pi'' \rightsquigarrow \Pi'}{\overline{D} \vdash s_1; s_2 : \Pi \rightsquigarrow \Pi'}$$

[Call]
$$\frac{m(\overline{z'})\ \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K\ \{\ \ldots\ \texttt{return}\ r\ \} \in \overline{D} \qquad \theta = [\overline{z' := z}, \texttt{this} := y, r := x] \qquad \forall p \in \theta(P).\ \Pi \vdash p}{\overline{D} \vdash x\ \texttt{=}\ y.m(\overline{z}) : \Pi \rightsquigarrow (\Pi \setminus K)[x := w] \cup \theta(Q)}$$

[Method]
$$\frac{\begin{array}{c} m \text{ is unique in } \overline{D} \qquad s \text{ does not mutate } \texttt{this}, \overline{x} \\ K \text{ contains all fields that may be modified while evaluating } s \\ K \text{ contains } \textsc{All} \text{ if a lock may be released while evaluating } s \\ \mathrm{FV}(P) \subseteq \{\texttt{this}, \overline{x}\} \qquad \mathrm{FV}(Q) \subseteq \{\texttt{this}, \overline{x}, r\} \\ \overline{D} \vdash s : P \rightsquigarrow Q' \qquad Q' \sqsubseteq Q \end{array}}{\overline{D} \vdash m(\overline{x})\ \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K\ \texttt{\{}\ \ s; \texttt{return}\ r\ \texttt{\}}}$$

$\boxed{\overline{D} \vdash meth \quad \vdash \overline{D} \quad \vdash \overline{D}\ s_1 \| \ldots \| s_n}$

[Class]
$$\frac{\forall meth \in \overline{meth}.\ \overline{D} \vdash meth}{\overline{D} \vdash \texttt{class}\ C\ \texttt{\{}\ \overline{f}\ \ \overline{meth}\ \texttt{\}}}$$

[Declarations]
$$\frac{\forall D \in \overline{D}.\ \overline{D} \vdash D}{\vdash \overline{D}}$$

[Program]
$$\frac{\vdash \overline{D} \qquad \forall i \in 1..n.\ \overline{D} \vdash s_i : \emptyset \rightsquigarrow \Pi}{\vdash \overline{D}\ s_1 \| \ldots \| s_n}$$

$\boxed{\Pi \setminus K,\ \Pi \setminus \kappa}$

$$\Pi \setminus \{\kappa_1, \ldots, \kappa_n\} = \Pi \setminus \kappa_1 \setminus \ldots \setminus \kappa_n$$
$$\Pi \setminus f = \{\ p \in \Pi\ \} \cup \{\ be \in \Pi\ \} \cup \{\ (y = p) \in \Pi\ \mid\ p \neq x.f\ \}$$
$$\Pi \setminus \textsc{All} = \{\ be \in \Pi\ \}$$

Fig. 4. Analysis (We assume w is fresh in all rules.)

where the *PathSet* P denotes the paths that must be in the context at any call site to the method, which enables inter-procedural check elimination. The PathSet Q denotes paths that are available at the end of the method body, and the *ModifiesSet* K denotes fields that may be modified by the method body. (See Figure 3.) The ModifiesSet K also contains the special token All if the method contains a lock release or other span-ending operation.

3.1 Type System

The core of our analysis is the set of rules in Figure 4 defining the judgment $\overline{D} \vdash s : \Pi \rightsquigarrow \Pi'$, where the context Π denotes available paths and constraints

that hold in the pre-state of statement s, and context Π' similarly characterizes the post-state of s. The definitions $\overline{D}$ are included to provide access to class declarations. The complete set of statement typing rules, as well as rules for additional judgments to check declarations and programs, appear in Figure 4. We describe the most salient rules below.

[Assign] Assigning to a local variable x may affect constraints or paths containing x. Simply removing all such elements mentioning x from Π would be overly conservative. Instead, we introduce a class of *Skolem variables* that are implicitly existentially quantified. The post-context for a statement x `=` e from a pre-context Π is then computed to be $\Pi[x := w] \cup \{x = e[x := w]\}$, where the fresh implicitly existentially quantified variable w captures the pre-assignment value of x. (The substitution $\Pi[x := w]$ replaces all occurrences of x with w in Π, and similarly for $e[x := w]$.)

To illustrate this rule, consider the code fragment to the right. The context prior to the assignment to `z` contains `a` = `z.f` and `b` = `z.f`, which implies `a` and `b` are aliases. After the assignment to `z`, the context will contain `a` = w`.f` and `b` = w`.f` for some fresh w. Thus we may still prove `a` and `b` are aliases and the last access is redundant, even though `a` and `b` are no longer equal to `z.f`.

```
a = z.f^Check;
b = z.f^NoCheck;
z = y;
t1 = a.g^Check;
t2 = b.g^NoCheck;
```

Note that this rule adds the equality $x = e[x := w]$ to Π, where e may be any expression. The most useful equality constraints for reasoning about field accesses have the form $x = y$, but we will see in Section 4 that constraints over more complex scalar expressions are crucial for reasoning about array accesses.

[Read] For the read statement x `=` $y.f^k$, we verify that $y.f$ has already been accessed in the current span if k = `NoCheck`. To do this, we could simply require that $y.f \in \Pi$. However, this syntactic notion of membership does not take into account aliasing or other information. Thus we introduce the *context implication* relation $\Pi \vdash y.f$ to indicate that the elements in context Π imply that $y.f$ has been accessed, perhaps via an aliased name:

$$\Pi \vdash y.f \quad \text{iff} \quad \Pi \vdash z = y \wedge z.f \in \Pi$$

Any sound decision procedure may be used to implement the relation $\Pi \vdash z = y$. In our implementation, we leverage the SMT solver Z3 [11], making sure that the translation into Z3's input language is appropriately conservative. Below, we use a generalized context implication relation $\Pi \vdash \pi$ to express that the context Π implies the context item π, which may be a path, alias constraint, or a boolean constraint.

The substitution $y.f[x := w]$ in this rule ensures that the proper elements are added to post-context Π' in the case where $y = x$.

[Write] For the write statement $y.f^k$ `=` x, we verify that $y.f$ has already been accessed in the current span if k = `NoCheck`. To compute the post-context Π', we remove all equalities referring to the f field of an object via the

operation $\Pi \setminus f$ defined in Figure 4. After the assignment, we include both the equality $x = y.f$ and the available path $y.f$ in Π'.

[If] The rule for conditionals merges the post-contexts of the *then* and *else* branches via the meet operator

$$\Pi_1 \sqcap \Pi_2 \quad = \quad \{\ \pi \in \Pi_1 \cup \Pi_2 \ \mid\ \Pi_1 \vdash \pi \text{ and } \Pi_2 \vdash \pi\ \}$$

This operator computes the largest set of facts in $\Pi_1 \cup \Pi_2$ that are provable from both Π_1 and Π_2. Those contexts may refer to different Skolem variables, but this rule does not attempt to unify them in any way. Thus, any π in $\Pi_1 \sqcap \Pi_2$ will only refer to Skolem variables appearing in both Π_1 and Π_2.

[While] The rule for loops identifies an appropriate loop invariant context Π_{inv} that is implied by the pre-state of the loop and also preserved by each iteration of the loop body. Implication between contexts is defined as

$$\Pi_1 \sqsubseteq \Pi_2 \quad \text{iff} \quad \forall \pi \in \Pi_2.\ \Pi_1 \vdash \pi$$

Our implementation uses a form of Cartesian predicate abstraction [20,22] to heuristically compute appropriate loop invariants, as described in Section 6. It synthesizes initial candidate invariants based on loop induction variable and bounds information extracted from the code and pattern matching for common idioms.

[Call] This rule for a method call x `=` $y.m(z_{1..n})$ first ensures that there is a corresponding method definition

$$m(\overline{z'})\ \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K\ \{\ s; \texttt{return}\ r\ \}$$

and creates a substitution θ to map `this`, the formal parameters $\overline{z'}$, and the result variable r to the corresponding variables y, $\overline{z}$, and x at the call site. The PathSet precondition P describes paths that must be available at call sites, and this rule checks that each path $p \in \theta(P)$ is entailed by Π.
The ModifiesSet K contains fields assigned to while evaluating the body of m (either directly, or indirectly while evaluating a nested method call), as well as a special token All if m performs a release or other span-ending operation. (That requirement is enforced in [Method].) The operation $\Pi \setminus K$ removes any invalidated constraints from the context Π. We also replace occurrences of x in Π as in the previous rules and add the paths from the method's post-context Q, yielding the final context $\Pi' = (\Pi \setminus K)[x := w]\ \cup\ \theta(Q)$.

[Acq] Acquires do not change the current context.

[Rel] Once a lock is released, we cannot assume any location is race-free or any aliasing constraint still holds. Thus, we remove all aliasing information and available paths from the context, leaving just the boolean expression constraints, which only refer to local variables.

3.2 Correctness

We state our main soundness theorem for an appropriate extensions of our type system to run-time states, denoted $\overline{D}, \alpha \vdash \Sigma$. (Please see [19] for the full formal

development). This judgment ensures that the run-time state is consistent with the statically-computed context for each thread. The judgment includes the trace α to provide information about previous accesses performed within currently active spans. Any verified program state Σ generates only well formed traces α:

Theorem 2 (Soundness). *If* $\overline{D}; \epsilon \vdash \Sigma$ *and* $\Sigma \longrightarrow^{\alpha} \Sigma'$*, then* α *is well-formed.*

Thus, by Theorem 2, a race detector can safely ignore `NoCheck` accesses in any verified program, with no loss in precision.

4 Arrays

In order to better optimize array-intensive programs, we extend our analysis to support universally quantified paths, as illustrated by the following code fragment to clear an array, as well as computed analysis contexts:

```
i = 0;
// Π : i = 0
while (i < a.length) {
  // Π : ∀(j ∈ 0 to i). a[j], i < a.length
  a[i]^Check = 0;
  // Π : ∀(j ∈ 0 to i). a[j], a[i], i < a.length
  i = i + 1;
  // Π : ∀(j ∈ 0 to i'). a[j], a[i], i = i' + 1, i' < a.length
}
// Π : ∀(j ∈ 0 to a.length). a[j], i ≥ a.length
```

We wish to infer that after the loop terminates, all array elements have been accessed, which we capture as the *universally quantified path*

$$\forall j \in (0 \texttt{ to a.length}).\ \texttt{a}[j]$$

stating that array elements `a[0]`, `a[1]`, ..., `a[a.length−1]` have been accessed. Verifying this loop post-condition naturally requires a corresponding loop invariant path

$$\forall j \in (0 \texttt{ to i}).\ \texttt{a}[j]$$

and also a richer constraint language to capture relevant invariants on indexed variables. For example, on entry to the loop, the equality $\texttt{i} = 0$ ensures that the loop invariant holds initially:

$$\{\ \texttt{i} = 0\ \} \Rightarrow \{\ \forall j \in (0 \texttt{ to i}).\ \texttt{a}[j]\ \}$$

Moreover, the invariant holds on the loop's back edge due to the following implication, where the Skolem variable i' refers to the value of `i` before the increment:

$$\{\ \forall j \in (0 \texttt{ to } i').\ \texttt{a}[j], \texttt{a}[i'], \texttt{i} = i' + 1\ \} \ \Rightarrow\ \{\ \forall j \in (0 \texttt{ to i}).\ \texttt{a}[j]\ \}$$

$$
\begin{array}{rcll}
s & \in & \mathit{Stmt} & ::= \ \ldots \ \mid \ x \texttt{ = new } C\texttt{[}z\texttt{]} \ \mid \ x \texttt{ = } y\texttt{[}z\texttt{]}^k \ \mid \ y\texttt{[}z\texttt{]}^k \texttt{ = } x \\
p & \in & \mathit{Path} & ::= \ x.f \ \mid \ x\texttt{[}e\texttt{]} \\
qp & \in & \mathit{QuantifiedPath} & ::= \ p \ \mid \ \forall i \in r.\ qp \\
r & \in & \mathit{StridedRange} & ::= \ e \texttt{ to } e \texttt{ by } e \\
\pi & \in & \mathit{ContextItem} & ::= \ qp \ \mid \ c \ \mid \ be
\end{array}
$$

$$\boxed{\overline{D} \vdash s : \Pi \rightsquigarrow \Pi'}$$

[Array Read]

$$\frac{k = \texttt{NoCheck} \ \Rightarrow \ \Pi \vdash y\texttt{[}z\texttt{]} \qquad p = (y\texttt{[}z\texttt{]})[x := w] \qquad \Pi' = \Pi[x := w] \cup \{x = p, p\}}{\overline{D} \vdash x \texttt{ = } y\texttt{[}z\texttt{]}^k : \Pi \rightsquigarrow \Pi'}$$

[Array Write]

$$\frac{k = \texttt{NoCheck} \ \Rightarrow \ \Pi \vdash y\texttt{[}z\texttt{]} \qquad \Pi' = (\Pi \setminus \textsc{Array}) \cup \{x = y\texttt{[}z\texttt{]}, y\texttt{[}z\texttt{]}\}}{\overline{D} \vdash y\texttt{[}z\texttt{]}^k \texttt{ = } x : \Pi \rightsquigarrow \Pi'}$$

Fig. 5. RedJava Extensions to Support Arrays

Finally, on loop termination, we verify:

$$\{\ \forall j \in (0 \texttt{ to i}).\ \texttt{a[}j\texttt{]}, \texttt{i} \geq \texttt{a.length}\ \} \ \Rightarrow \ \{\ \forall j \in (0 \texttt{ to a.length}).\ \texttt{a[}j\texttt{]}\ \}$$

Figure 5 formalizes the extended language of quantified paths and type rules used by our analysis. Paths now include array accesses $x\texttt{[}z\texttt{]}$ as well as field accesses. Paths may also include an enclosing quantification ($\forall i \in r.\ \bullet$) over a strided range r of the form "e_{start} `to` e_{end} `by` e_{step}", which represents the set of indices $\{\ i \in [e_{start}, e_{end}) \mid (i - e_{start}) \mod e_{step} = 0\ \}$. (We abbreviate the strided range "e_{start} `to` e_{end} `by` 1" as "e_{start} `to` e_{end}".)

The rule [Array Write] for an array assignment $y\texttt{[}z\texttt{]} \texttt{ = } x$ uses the operation $\Pi \setminus \textsc{Array}$, defined below, to remove from Π any information dependent on array values. We also include Array as a possible ModifiesSet component κ in method specifications.

$$
\begin{array}{rcl}
\kappa & ::= & f \ \mid \ \textsc{All} \ \mid \ \textsc{Array} \\
\Pi \setminus \textsc{Array} & = & \{\ p \in \Pi\ \} \ \cup \ \{\ be \in \Pi\ \} \ \cup \ \{\ (x = p) \in \Pi \ \mid \ p \neq y\texttt{[}e\texttt{]}\ \}
\end{array}
$$

This treatment of array writes is quite coarse since an assignment to any array eliminates facts about all arrays in the program. In practice, we use a type-based rule that, when an array of type C is modified, eliminates only facts dependent on values stored in arrays of type C. This refinement works well in practice, but further refinements based on more precise points-to information could be used in cases where it proves insufficient.

5 Shadow Proxies

Identifying redundant checks can improve the performance of dynamic race detectors, but it does not directly reduce the memory overhead of keeping analysis

(or *shadow*) state for each memory location. Various race detectors [32, 43] have explored using one shadow state per object or array, but this approach may produce spurious warnings.

We now explore how to reduce the number of shadow states via static analysis, while still preserving precision. Our approach is based on the observation that accesses to different memory locations are often correlated. For example, an object ρ of type C with fields f and g may be used in such a way that whenever $\rho.g$ is accessed, $\rho.f$ will also be accessed *in the same span*. Thus any data race on $\rho.g$ naturally implies there is a race on $\rho.f$. In this case, we say that $\rho.f$ is a *proxy location* for $\rho.g$, and we say that f is a *proxy field* for g, written $f \succ g$, if that proxy relationship holds for all C objects.

If $f \succ g$, then a race checker may forego allocating shadow state for g fields and ignore all accesses to them while still providing the following guarantees for all objects ρ:

- If no races are found on $\rho.f$, then both $\rho.f$ and $\rho.g$ are race-free.
- No races will be reported on $\rho.g$, since those accesses are not checked.
- A detected race on $\rho.f$ implies that there *is* a race on $\rho.f$ and that there *may be* a race on $\rho.g$.

Note that if both $f \succ g$ and $g \succ f$ (*i.e.*, f and g are always accessed together), then the third guarantee may be strengthened to:

- A detected race on $\rho.f$ implies that there are races on *both* $\rho.f$ and $\rho.g$.

However, the weaker asymmetric requirement $f \succ g$ enables more check elimination and memory footprint reduction while still providing the useful correctness guarantee of always detecting a data race on any trace containing one.

Although space limitations prevent a full discussion, we summarize type system extensions for identifying proxy fields in the rest of this section. The key typing requirement is that $f \succ g$ if and only if, for each access instruction x `=` $y.g^k$ or $y.g^k$ `=` x, either

- $\Pi \vdash y.f$, where Π is the instruction's computed context; or
- $\Pi' \vdash y.f$, where Π' is the computed context for some instruction in the same span that post-dominates the access to g.

The first clause captures cases when f is accessed before g, and the second captures the converse. If these requirements hold, all race checks for g are redundant. In our Java implementation, we compute the post-dominator relation assuming all unchecked exceptions, such as `NullPointerException`s, are errors in the source code and guarantee no loss in precision only on error-free traces.

Array entries may also have proxies. Location ρ`[`i`]` is a proxy location for ρ`[`j`]` if ρ`[`i`]` is accessed in every span accessing ρ`[`j`]`. We say that i is a *proxy index* for j if that requirement holds for all array accesses in a program. Distinguishing arrays by allocation site (via, for example, a standard points-to analysis) enables a more precise definition: $i \succ_l j$ if i is a proxy index for j for all arrays allocated at source location l. The key typing requirement for arrays is that $i \succ_l j$ if and

only if, for each access instruction x `=` y`[`z`]`^{k} or y`[`z`]`^{k} `=` x such that y may be an array allocated at source location l and z's value may be j, either

- $\Pi \vdash y$`[`i`]`, where Π is the instruction's computed context; or
- $\Pi' \vdash y$`[`i`]`, where Π' is the computed context for some instruction in the same span that post-dominates the array access.

Given this definition, the race check on an access y`[`z`]` is redundant if, for every possible value j for z and allocation site l for y, there exists an i such that $i \succ_l j$.

Typically, shadow proxies for all indices of an array can be summarized quite succinctly. For example, the relation $\forall i.\ (i \text{ div } 4) * 4 \succ_l i$ indicates that all arrays allocated at l are divided into chunks of four elements, where the first index in each chunk is a proxy for the other three indexes in the chunk. Any array access instruction guaranteed not to touch one of the proxies may be tagged with `NoCheck`. Moreover, a race detector can use the shadow location maintained for each chunk's proxy when checking accesses to any elements in the chunk, reducing the number of required shadow locations for an array of n elements from n to $n/4$. Similar reductions in checking and memory requirements are possible for many different proxy relations, several of which are described below.

The shadow proxy analysis requires information about each span. However, one may be able to identify many proxies without examining all spans: for example, proxies among private fields can be found by examining only a single class at a time. We could also augment our system with simple "proxy specifications" on classes, which could then be subsequently verified by a modular analysis. We hope to explore these items in more detail in the future.

6 Implementation

We have implemented our analysis for the full Java language as part of a tool called RedCard. This analysis reads in bytecode programs and labels each field and array access as either `NoCheck` or `Check`. It outputs a list of the accesses marked `NoCheck`, and that list may then be used to optimize any dynamic race detector. Transforming our type system to the full Java language and an inference algorithm for bytecode is mostly straightforward. We describe the most interesting implementation details below.

We implemented RedCard in the WALA analysis infrastructure [41], which represents method bodies as control flow graphs over instructions in SSA form. The tool analyzes all methods appearing in a call graph built by WALA using 0-CFA. RedCard implements the relation $\Pi \vdash \pi$ via the Z3 SMT Solver [11].

For each instruction in each method, we compute a context Π via dataflow analysis. All contexts begin as the special context Top, where $\Pi \sqcap \text{Top} = \Pi$ for all contexts Π. The analysis then uses a meet operator and transfer functions based on the system presented in Section 3.1 to compute the maximal fixed point solution for the contexts for each instruction. RedCard handles all basic synchronization operations present in Java, including locks, volatile variables, fork/join, wait/notify, and so on. RedCard then identifies each memory access instruction for which its context implies a check is redundant.

Intra- and Inter-Procedural Modes. RedCard processes methods in one of two modes: an *intra-procedural mode* in which RedCard assumes the most conservative specification "`requires { } ensures { } modifies { All }`" for all methods and proceeds to infer redundant checks via an intra-procedural dataflow analysis; or an *inter-procedural mode* in which RedCard uses a context-insensitive inter-procedural dataflow analysis to infer both method specifications and redundant checks. The inter-procedural mode also uses a class hierarchy analysis to reason about method resolution. This mode yields better results when precise specifications are not available, but it is also a more complex and compute-intensive analysis.

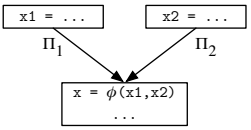
Distinguishing Reads and Writes. Up to this point, our analysis has not distinguished reads and writes. However, it is necessary to do so in practice since precise dynamic race detectors treat them differently. Specifically, two concurrent accesses are considered conflicting only when at least one of them is a write. Given this distinction, RedCard uses the following rules to determine whether a check is redundant:

- A dynamic check on a **write** is redundant if there is a previous write to the same location in the current span.
- A dynamic check on a **read** is redundant if there is a previous read *or write* to the same location in the current span.

Thus, we record in Π whether each access p is a read or a write, and we adjust the definition of redundancy to match the above. For simplicity, we continue to treat all accesses uniformly in the examples below.

Libraries. Since we evaluate RedCard when used in conjunction with FastTrack, RedCard follows FastTrack's treatment of libraries: Fields of library classes are not checked for races, and synchronization operations internal to libraries are assumed to not be used to protect any of the target's data and are ignored. However, several key library methods from `java.lang.Object` and `java.lang.Thread`, such as `Object.notify` and `Thread.start`, are treated specially as synchronizing operations. These assumptions may cause FastTrack to report false alarms (since necessary synchronization within libraries may be ignored), but we have never observed false alarms for the benchmarks studied. RedCard treats synchronization in libraries in exactly the same way, thereby leaving the completeness/soundness guarantees of FastTrack unchanged. The programmer can also provide more precise library module specifications or fully analyze specific library classes via RedCard command-line options.

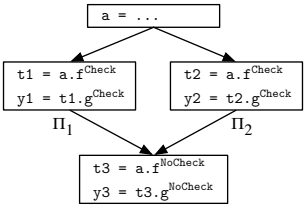
ϕ-functions. WALA's SSA representation makes some aspects of the dataflow analysis much easier since local variables are immutable. However, SSA does have ϕ-functions for merging multiple definitions into a single variable at meet-points in the CFG, as in the following example, which also shows how we compute Π for the entry to the block containing a ϕ-function:



$$\begin{aligned} \Pi \;&=\; (\Pi_1[\mathtt{x} := w_1] \cup \{\mathtt{x} = \mathtt{x1}\}) \\ &\sqcap\; (\Pi_2[\mathtt{x} := w_2] \cup \{\mathtt{x} = \mathtt{x2}\}) \end{aligned}$$

In essence, we equate `x` with the appropriate variable in the incoming contexts Π_1 and Π_2. Contexts propagated along back edges introduce one additional complexity. For example, if Π_2 is the context propagated back to the top of a loop containing the definition of `x`, then Π_2 may already include references to `x`. Thus, we replace all occurrences of `x` in Π_2 with a fresh, Skolem variable w_2, and similarly modify Π_1, prior to adding the equalities and computing the meet of the resulting contexts.

Path Expansion. Our previously defined meet operator can be overly coarse at meet-points. To illustrate why, consider the following program and contexts:



$$\Pi_1 = \{\ \mathtt{a.f},\ \mathtt{t1} = \mathtt{a.f},\ \mathtt{t1.g},\ \mathtt{y1} = \mathtt{t1.g}\ \}$$

$$\Pi_2 = \{\ \mathtt{a.f},\ \mathtt{t2} = \mathtt{a.f},\ \mathtt{t2.g},\ \mathtt{y2} = \mathtt{t2.g}\ \}$$

$$\begin{aligned} \Pi_1 \sqcap \Pi_2 &= \{\ \pi \in \Pi_1 \cup \Pi_2 \ \mid\ \Pi_1 \vdash \pi \text{ and } \Pi_2 \vdash \pi\ \} \\ &= \{\mathtt{a.f}\} \end{aligned}$$

Both branches access `a.f.g`. Thus, the check on `y3 = t3.g` is redundant. However, we cannot prove that because the context for the start of the final block would be $\Pi_1 \sqcap \Pi_2 = \{\mathtt{a.f}\}$. In essence, the meet operation loses information about available paths because incoming contexts may encode aliasing via different local variables. To reason more precisely about such situations at meet-points, we permit paths to include more than one field or array reference, and RedCard expands each access path in Π_1 and Π_2, using their respective aliasing constraints, until the access paths refer only to variables defined by instructions common to both control-flow paths leading to the meet point (*i.e.*, that dominate the meet-point). In Π_1 above, `t1.g` and `t1 = a.f` are combined to obtain `a.f.g`. The same path is similarly derived from Π_2. These expanded paths are added back into Π_1 and Π_2 before $\sqcap$ is applied, yielding

$$(\Pi_1 \cup \{\mathtt{a.f.g}\}) \sqcap (\Pi_2 \cup \{\mathtt{a.f.g}\}) = \{\ \mathtt{a.f},\ \mathtt{a.f.g}\ \}$$

which does allow us to conclude that checking for races on `y3 = t3.g` is redundant. Supporting longer paths requires changes to PathSet and ModifiesSet operations, but it is mostly straightforward. Our implementation limits paths to contain at most four field or array references to ensure termination.

Loop Invariants. RedCard infers loop invariants with a specialized form of Cartesian predicate abstraction [20, 22] on both access paths and constraints. More specifically, we compute the loop invariant context Π_{inv} in rule [while] by

first heuristically generating a conjectured context $\Pi_{heuristic}$ and then repeatedly analyzing the loop body to infer the maximal $\Pi_{inv} \subseteq \Pi_{heuristic}$ that is a valid loop invariant. Our heuristics conjecture invariants based, in part, on the context that first flows to the loop head, inferred loop induction variables, and pattern matching for common idioms. Unusual array access patterns, complex index computations, and irreducible flow graphs may be problematic for our approach, but we found it to work quite well in practice for most loops, including nested loops iterating over multi-dimensional arrays in various ways.

Shadow Proxies. After computing the contexts for every program point, RedCard infers field shadow proxies by conjecturing that each field is a proxy for every other field and then refuting the conjectures that do not hold.

To identify array shadow proxies, RedCard first conjectures a set of possible index relations that hold for all arrays allocated at each allocation site l. These include relations of the form "$\forall i.\ F(i) \succ_l i$" where $F(i)$ is one of the following:

$$\begin{array}{lll} F(i) = 0 & F(i) = (i \text{ div } 4) * 4 & F(i) = i \bmod 4 \\ F(i) = 1 & F(i) = (i \text{ div } 8) * 8 & F(i) = i \bmod 8 \end{array}$$

The first column characterizes whole arrays proxied by a single index, the middle column characterizes arrays divided into chunks where an index is proxied by the first index in its chunk, and the right column characterizes arrays that are always traversed by a fixed stride (*e.g.*, every fourth element is touched). RedCard then refutes the proxy relations for arrays that do not hold. We believe further improvements could be made by expanding this set of conjectures and refining our analysis to reason about more sophisticated patterns.

Code Transformations. A variety of code transformations significantly improve the number of statically identifiable redundant checks. One particularly useful transformation is loop unrolling [8,38]. For example, the following loop (on the left) performs N run-time race checks on `a.f`. Assuming `m` does not modify `a.f`, all of these checks are redundant except the first. Thus the semantically-equivalent version on the right performs only one check.

```
for (i = 0; i < N; i++) {
  a.f^Check.m();
}
```

```
i = 0;
if (i < N) {
  a.f^Check.m();
  for (i = 1; i < N; i++) { a.f^NoCheck.m(); }
}
```

RedCard currently implements loop unrolling via a source-to-source translation step occurring prior to the dataflow analysis. Another transformation we hope to explore in the future is method specialization based on available paths at different call sites. This optimization is synergistic with loop unrolling, since a method called inside the loop body can be specialized to two versions: one for the first iteration outside the loop, and one version for inside the loop where fewer checks may be necessary.

7 Evaluation

We demonstrate the effectiveness of RedCard by evaluating its ability to eliminate redundant checks in a variety of benchmarks. We first describe our experimental framework and analysis time, then show the percentage of access checks removed by RedCard for each program, and finally demonstrate how eliminating checks improves the performance of the FastTrack race detector.

Benchmark Configuration. We performed experiments on the following benchmarks: `elevator`, a discrete event simulator for elevators [38]; `hedc`, a tool to access astrophysics data from Web sources [38]; `tsp`, a Traveling Salesman Problem solver [38]; `mtrt`, a multithreaded ray-tracing program from the SPEC JVM98 benchmark suite [36]; `jbb`, the SPEC JBB2000 business object simulator [36]; `crypt`, `lufact`, `sparse`, `series`, `sor`, `moldyn`, `montecarlo`, and `raytracer` from the Java Grande benchmark suite [25]; the `colt` scientific computing library [6]; the `raja` ray tracer [21]; and and `philo`, a dining philosophers simulation [14]. We configured the Java Grande benchmarks to use four worker threads and the largest data set provided. Table 1 shows statistics for both the "Original" programs and "Unrolled" variants in which RedCard transformed all inner-most loops as described in Section 6. All experiments were performed on an Apple Mac Pro with dual 3GHz quad-core Pentium Xeon processors and 12GB of memory running Sun's Java HotSpot 64-bit Server VM version 1.6.0. Access counts and running times are the average of ten executions.

RedCard Static Analysis Time. The time required by RedCard to process the "Original" benchmarks was on average 18 seconds per thousand lines of code. Approximately 40% of this time was spent loading class files and building the internal data structures used by the WALA analysis engine. The remaining time was primarily consumed by the RedCard dataflow analysis, which has not been optimized. We believe substantial speed ups could be achieved by refactoring and refining our core data structures. Typically, less than 15% of the total running time was spent solving Z3 queries, indicating that the SMT solver was not a bottleneck. The more complex control flow graphs in the "Unrolled" versions led to an average processing time of 24 seconds per thousand lines of code.

Analysis time increased by roughly 50% when RedCard was configured to identify field and array proxies. Our prototype conjectures a large set of possible field and array proxies, and then checks the validity of each conjecture independently. Replacing this approach by a more efficient algorithm that processes multiple conjectures simultaneously would eliminate much of this overhead.

Although we have not optimized RedCard for speed, the initial results show that it is no more expensive than existing whole-program techniques. For example, the geometric mean of the time required to process each program in the JavaGrande suite was 101 sec. for Chord and 55 sec. for RedCard. These tools are built on top of frameworks (Soot / WALA) with different performance characteristics, and a more consistent implementation and analysis of them would be required to draw definitive conclusions.

Table 1. Percentage of run-time race checks eliminated under different configurations

Program	Size (lines)	Accesses (Millions)	% Accesses Checked					
			Original			Unrolled		
			DynamicallyNonRedundant	RedCard	RC+Proxy	DynamicallyNonRedundant	RedCard	RC+Proxy
colt	111,162	1,102.42	99.6	99.9	99.6	99.3	99.9	99.9
crypt	1,255	2,150.0	56.4	59.2	46.7	40.7	44.6	43.5
lufact	1,627	7,531.07	99.1	99.4	99.4	99.1	99.3	99.3
moldyn	1,409	30,586.08	53.1	59.5	27.5	26.3	27.7	14.8
montecarlo	3,669	2,713.35	2.3	43.0	34.8	2.3	28.8	24.6
mtrt	11,315	24.23	40.2	77.8	77.0	40.2	77.7	76.8
raja	12,027	5.39	55.6	96.8	69.9	55.6	96.8	69.9
raytracer	1,971	39,560.09	51.5	86.2	34.0	48.4	83.2	31.0
sparse	868	7,244.5	83.0	90.5	56.1	34.6	42.1	42.1
series	967	4.0	75.0	83.3	66.7	0.1	33.3	33.3
sor	1,017	2,417.42	83.3	83.5	83.5	66.3	82.8	82.8
tsp	706	820.71	62.9	92.8	86.5	61.3	75.6	69.5
elevator	2,840	0.02	69.2	80.0	66.5	58.5	69.5	61.2
philo	86	<0.01	64.5	67.3	59.4	54.5	59.0	52.2
hedc	24,932	0.05	5.9	94.6	94.5	2.3	93.5	93.4
jbb	45,943	1,068.74	75.8	84.2	78.3	68.4	75.4	69.3
Geo. Mean			46.7	79.3	63.1	24.6	62.6	53.4

Redundant Check Elimination. Table 1 shows the number of accesses performed by each program. A precise race detector would need to check 100% of those accesses. That table also shows how many of those checks could be eliminated under three scenarios:

Dynamically NonRedundant: To gauge how well the inherently conservative RedCard analysis performs, we first estimate the "optimal" set of instructions that could be annotated as `NoCheck` by examining run-time access histories. Specifically, we ran each program and identified access instructions that only referenced locations already accessed in the current span. We labeled those instructions as `NoCheck` and all others as `Check`. This may over-estimate the number of removable checks due to coverage limitations but seems a reasonable approximation for general comparisons.

RedCard: This column reports the percentage of run-time accesses corresponding to checks that the RedCard static analysis labeled as `Check`.

RC+Proxy: This column also uses RedCard to label accesses, but reasons about shadow proxies for fields and arrays as well.

We show the number of accesses requiring run-time race checks for these scenarios as a percentage of the total number accesses. For the "Original" programs, RedCard reduced the geometric mean of accesses checked to 79.3% of the total

number of accesses. While still higher than the DYNAMICALLY NONREDUNDANT estimate of 46.7%, it does remove a substantial amount, and for a number of programs, REDCARD was close to the estimated optimal.

Inferring shadow proxies proved particularly useful for a number of programs, as shown in the RC+PROXY column. For example, in `raytracer`, REDCARD recognized that the `x`, `y`, and `z` fields of a heavily-used 3-D point class were proxies, leading to a reduction in the number of checks performed from 86% to 34%. Other programs, such as `crypt`, contained large arrays for which proxy relations could be computed. Overall, REDCARD with proxy inference roughly cut in half the number of run-time race checks.[3] Switching to the *intra-procedural mode* still provided significant benefit, reducing the checking rate to about 83% for REDCARD, and to about 70% for RC+PROXY.

For the "Unrolled" programs, REDCARD performed somewhat better and reduced the checking rate to roughly 63% for REDCARD and to 53% for RC+PROXY.

Overall, these results are quite promising. We note that there is high variance in how well REDCARD performs relative to the estimated optimal. In those cases where REDCARD performed poorly, imprecisions in aliasing information, array index computations, or loop invariant conjectures made it unable to verify that commonly exercised instructions could be tagged as `NoCheck`. Improving these items would enable REDCARD to reason about more subtle code.

Run-time Performance. We now examine how REDCARD improves the run-time performance of the FASTTRACK precise dynamic race detector [18]. FASTTRACK loads unmodified Java class files and instruments synchronization and memory operations to generate an event stream. It processes those events as the program executes.FASTTRACK tracks happens-before orderings between synchronization and memory accesses using an adaptive epoch-based representation.

FASTTRACK's representation of the happens-before relation is quite time and space efficient compared to other precise detectors, but it must check every access and maintain shadow state for every location. Table 2 shows the base running time of our target programs, as well as the slowdown incurred by FASTTRACK. When running FASTTRACK and our other analysis tools, all classes loaded by the benchmark programs were instrumented, except those from the standard Java libraries. The timing measurements include the time to load, instrument, and execute the target program startup time. On average, using FASTTRACK led to a slowdown of 7.5x. Memory accesses are far more common than synchronization operations in Java programs, and the vast majority of FASTTRACK's overhead is caused by monitoring field and array accesses.

The fourth column in Table 2 shows the slowdown of FASTTRACK when used in conjunction with REDCARD. That redundancy analysis reduced FASTTRACK's average slowdown from 7.5x to 7.1x. Including shadow proxy analysis further reduced the average slowdown to 5.7x, thereby eliminating approximately 25% of FASTTRACK's run-time overhead, as computed as the ratio of the run-

[3] RC+PROXY may yield better results than DYNAMICALLY NONREDUNDANT because that estimate does not take possible proxy relationships into account.

Table 2. Performance results. Programs marked with * are not compute-bound and are excluded from the mean slowdowns.

Program	Thread Count	Base Time (sec)	Slowdown (x Base Time)						Shadow Locations	
			Original				Unrolled			
			FastTrack	RedCard	RC+Proxy	$\left(\frac{\text{RC+Proxy}}{\text{FastTrack}}\right)$	RedCard	RC+Proxy	Base Count (x 10^5)	RC+Proxy (% of Base)
colt	11	16.0	1.1	1.1	1.1	(0.96)	1.1	1.0	5.1	90.0
crypt	7	1.2	35.4	35.1	13.1	(0.37)	35.0	20.6	1,262.8	13.8
lufact	4	5.6	8.2	7.7	7.7	(0.94)	8.1	8.2	40.1	99.9
moldyn	4	7.5	10.5	7.9	4.7	(0.45)	12.8	6.8	4.7	56.2
montecarlo	4	5.8	9.1	8.2	8.3	(0.91)	8.2	8.0	1,825.2	100.0
mtrt	5	0.4	11.8	10.9	9.5	(0.81)	10.7	9.6	23.0	97.5
raja	2	0.4	6.8	7.2	6.4	(0.94)	7.1	6.5	9.6	59.0
raytracer	4	5.8	14.2	12.5	6.0	(0.42)	12.0	5.9	2,098.6	43.7
sparse	4	5.9	19.9	19.9	19.6	(0.99)	20.6	20.8	159.7	100.0
series	4	150.1	1.0	1.0	1.0	(1.00)	1.0	1.0	20.0	100.0
sor	4	2.3	6.2	6.1	5.9	(0.94)	6.5	6.7	40.0	100.0
tsp	5	0.6	7.3	7.1	7.1	(0.97)	11.1	10.8	1.6	100.0
elevator*	5	5.0	1.2	1.2	1.2	(1.00)	1.2	1.2	<0.1	86.5
philo*	6	9.3	0.6	0.5	0.6	(1.02)	0.6	0.7	<0.1	77.3
hedc*	6	4.7	1.3	1.3	1.3	(0.99)	1.4	1.3	0.4	100.0
jbb*	5	73.9	1.2	1.2	1.2	(1.00)	1.2	1.2	925.0	82.1
Geo. Mean			7.5	7.1	5.7	(0.76)	7.7	6.4		74.8

ning time of FastTrack configured to use RedCard with proxies and FastTrack's original running time in the parenthesized column. The improvements are not linear in the number of checks removed, due to other factors that can impact overall performance, such as lock contention on FastTrack's internal data structures and memory caching effects.

We had expected that the unrolled variants would exhibit greater performance improvements since RedCard eliminated more checks in them. However, they were actually about 15-20% slower on average. The primary cause of the slowdown appears to be the HotSpot JIT compilation engine, which was less able to optimize the unrolled code's larger methods and more complex control structures. We believe a tighter integration of the transformations, analysis, and JIT compilation strategy would mitigate these factors.

The last two columns in Table 2 evaluate RedCard's impact on the shadow memory maintained by FastTrack. The "Base Count" columns shows the number of distinct memory locations accessed by each program (and hence, the number of shadow locations FastTrack must allocate and maintain). A "shadow proxy-aware" FastTrack can avoid allocating shadows for all proxied memory locations, as described in Section 5. For the target programs, this led to a

roughly 25% average reduction in the number of allocated shadows. There is high variance among the benchmarks. Some programs in which no interesting proxy relationships were found show little change in the memory footprint, but others, such as `crypt`, `moldyn` and `raytracer`, show dramatic improvement, especially in how many shadow locations were allocated for tracking accesses to arrays.

Comparison to Chord. We also performed preliminary experiments to compare RedCard's ability to find redundancy to that of Chord, a sound static analysis for identifying accesses that may be involved in a data race [30]. Chord and other sound static analyses can often identify specific data structures or memory references guaranteed to be free of races, they may be used to verify or infer `NoCheck` annotations. Chord uses a collection of complex whole-program analyses to reason about reachability, aliasing, locking, and thread ordering. As such, Chord requires much more global reasoning than RedCard's *span-modular* redundancy analysis, particularly when used in the *intra-procedural mode*.

To gauge how well RedCard is able to reason about redundancy when compared to static analyses like Chord, we checked each benchmark program with Chord and labeled all accesses potentially involved in races as `Check`. All other accesses were labeled `NoCheck`. While the average reduction in checks across all benchmarks was roughly on par with RedCard, Chord's ability to reason about individual programs varied greatly. Less than 1% of the run-time checks were eliminated in some programs that use arrays heavily (`crypt`, `moldyn`, `raytracer`, `sparse`, `series`, `sor`). For other programs, greater than 99% of the run-time checks were eliminated (`lufact` and `mtrt`). We believe this bi-modal behavior is caused by several factors: Chord's handling of arrays is less precise than ours, and it seemed able to recognize that heavily used data structures in `lufact` and `mtrt` are thread-local and thus required no checking. Adding a thread-local analysis to RedCard may allow it to similarly remove many of these checks.

8 Related Work

Precise dynamic data race detectors typically represent the happens-before relation with *vector clocks* (VCs) [28], as in the Djit$^+$ race detector [32]. VCs are expensive to maintain, however. The FastTrack race detector uses an adaptive epoch-based representation to reduce the space and time requirements of VCs [18]. Other optimizations include dynamic escape analyses [8,38] or "accordion" vector clocks that reduce space overheads for programs with short-lived threads [10]. Different representations of the happens-before relation have also been explored [14]. Despite these improvements, the overhead of precise detectors can still be prohibitively high.

Similar notions of redundancy and release-free spans have been used in other settings. For example, the IFRit race detector uses the same insight about lock releases in its notion of interference-free regions [13], which were originally designed to facilitate compiler optimizations for race-free programs [12]. The IFRit race detector monitors execution and reports a data race when multiple concurrently executing interference-free regions access the same variable. IFRit is

faster than FastTrack, but is not precise and may miss some races. In contrast, RedCard can identify redundancies for any dynamic race detector, without introducing any false positives of false negatives. That is, any accesses identified as redundant can be skipped by any race detector without changing the guarantees provided by that detector.

Another study uses similar concepts to verify programmer-specified ownership policies [27], but that analysis is more restricted in its ability to reason about aliasing and function calls, and it requires programmers to specify when a thread has exclusive access to memory locations, rather than inferring it. Raman et al. developed a race condition algorithm for the very different context of structured parallelism, as in Cilk or X10 [33]. Their technique applies check hoisting and check elimination optimizations similar in spirit to RedCard to this domain. Their experimental results find fewer opportunities for optimization than RedCard, most likely due to either limitations in the precision in their analysis or the nature of programs written for such systems.

Many static analysis techniques for identifying races have also been explored, including systems based on types [1,3,23], model checking [7,29,42] and dataflow analysis [15], as well as scalable whole-program analyses [30,40]. While static race detection provides the potential to detect all race conditions over all program paths, decidability limitations imply that any sound static race detector may produce false alarms. As mentioned previously, these static analyses can often identify specific data structures or memory references guaranteed to be free of races and may thus be used to verify or infer `NoCheck` annotations. However, soundness of the static analysis is essential to avoid missing data races. Many of the mentioned static analyses are either unsound by design or unsound in their implementations to reduce the number of spurious warnings (see, *e.g.*, [1, 15]). Their focus on identifying race-free accesses rather than redundant race checks also lead to different design choices in terms of precision and scalability.

Gross *et al.* present a global static analysis to improve the precision and performance of a LockSet-based detector [38]. In contrast to RedCard's focus on redundant checks, their analysis, while it does eliminate some redundant checks on field accesses in a fairly restrictive way, is primarily designed to identify objects on which no races can occur. As such, their algorithm requires global aliasing information, as well as a static approximation of the happens-before graph for the whole program. Moreover, their reliance on an imprecise race detector leads their system to both miss races and report spurious warnings. They also do not support arrays.

Choi *et al.* present a different global analysis for removing run-time race checks for accesses guaranteed to be race-free [8]. Despite the primary focus on identifying race-free accesses, it does include elements closer in spirit to RedCard. In particular, the analysis eliminates some redundant checks via a simple intra-procedural analysis. However, their notion of redundancy is less general than ours. They cannot, for example, track redundancy across method calls, reason about array accesses, or model synchronization operations as precisely as

RedCard. In addition, their analysis is tailored for use with a variant of the imprecise LockSet algorithm. RedCard can be used to optimize any detector.

Various alias analyses have used notions similar to our available paths. For example, Fink *et al.* compute must and must-not aliases via access paths as part of an analysis to verify type-state protocols [16]. RedCard goes beyond that approach by supporting synchronization operations, a more precise model of arrays, and reasoning about race conditions. At first glance, RedCard seems to perform similar reasoning to analyses for redundant load elimination optimizations in compilers for concurrent languages, as in [4, 24, 39] for example. Our notion of redundancy, however, focuses on which locations have been accessed, and not on whether a value read from memory may be subsequently reused, as illustrated in Section 3. This leads to a more general notion of redundancy more amenable to analysis by tools specifically designed to reason about it.

Acknowledgments. This work was supported by NSF grants 0905650, 1116883 and 1116825. We thank James Wilcox for his assistance on the experiments.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for Java. TOPLAS 28(2), 207–255 (2006)
2. Adve, S.V., Hill, M.D., Miller, B.P., Netzer, R.H.B.: Detecting data races on weak memory systems. In: ISCA, pp. 234–243 (1991)
3. Agarwal, R., Stoller, S.D.: Type inference for parameterized race-free java. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 149–160. Springer, Heidelberg (2004)
4. Barik, R., Sarkar, V.: Interprocedural load elimination for dynamic optimization of parallel programs. In: PACT, pp. 41–52 (2009)
5. Boyapati, C., Rinard, M.: A parameterized type system for race-free Java programs. In: OOPSLA, pp. 56–69 (2001)
6. CERN. Colt 1.2.0., `http://dsd.lbl.gov/~hoschek/colt/`
7. Chamillard, A., Clarke, L.A., Avrunin, G.S.: An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst (1996)
8. Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridhara, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: PLDI (2002)
9. Choi, J.-D., Miller, B.P., Netzer, R.H.B.: Techniques for debugging parallel programs with flowback analysis. TOPLAS 13(4), 491–530 (1991)
10. Christiaens, M., Bosschere, K.D.: TRaDe: Data Race Detection for Java. In: International Conference on Computational Science, pp. 761–770 (2001)
11. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. Effinger-Dean, L., Boehm, H.-J., Chakrabarti, D.R., Joisha, P.G.: Extended sequential reasoning for data-race-free programs. In: MSPC, pp. 22–29 (2011)

13. Effinger-Dean, L., Lucia, B., Ceze, L., Grossman, D., Boehm, H.-J.: IFRit: interference-free regions for dynamic data-race detection. In: OOPSLA, pp. 467–484 (2012)
14. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A race and transaction-aware Java runtime. In: PLDI, pp. 245–255 (2007)
15. Engler, D.R., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: SOSP, pp. 237–252 (2003)
16. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. TOSEM 17(2) (2008)
17. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: PLDI, pp. 219–232 (2000)
18. Flanagan, C., Freund, S.N.: FastTrack: Efficient and precise dynamic race detection. Commun. ACM 53(11), 93–101 (2010)
19. Flanagan, C., Freund, S.N.: Redcard: Redundant check elimination for dynamic race detectors. Technical Report UCSC-SOE-13-05, UC Santa Cruz (2013)
20. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202 (2002)
21. Fleury, E., Sutre, G.: Raja, version 0.4.0-pre4 (2007), `http://raja.sourceforge.net/`
22. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
23. Grossman, D.: Type-safe multithreading in Cyclone. In: TLDI (2003)
24. Hosking, A.L., Nystrom, N., Whitlock, D., Cutts, Q.I., Diwan, A.: Partial redundancy elimination for access path expressions. Softw., Pract. Exper. 31(6), 577–600 (2001)
25. Java Grande Forum. Java Grande benchmark suite (2008), `http://www.javagrande.org`
26. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
27. Martin, J.-P., Hicks, M., Costa, M., Akritidis, P., Castro, M.: Dynamically checking ownership policies in concurrent C/C++ programs. In: POPL, pp. 457–470 (2010)
28. Mattern, F.: Virtual time and global states of distributed systems. In: Workshop on Parallel and Distributed Algorithms (1989)
29. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI (2008)
30. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI (2006)
31. Nishiyama, H.: Detecting data races using dynamic escape analysis based on read barrier. In: Virtual Machine Research and Technology Symposium, pp. 127–138 (2004)
32. Pozniansky, E., Schuster, A.: MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. Concurrency and Computation: Practice and Experience 19(3), 327–340 (2007)
33. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Efficient data race detection for async-finish parallelism. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 368–383. Springer, Heidelberg (2010)
34. Ronsse, M., Bosschere, K.D.: RecPlay: A fully integrated practical record/replay system. TCS 17(2), 133–152 (1999)
35. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multi-threaded programs. TOCS 15(4), 391–411 (1997)

36. Standard Performance Evaluation Corporation. SPEC benchmarks (2003), http://www.spec.org/
37. von Praun, C., Gross, T.: Object race detection. In: OOPSLA, pp. 70–82 (2001)
38. von Praun, C., Gross, T.: Static conflict analysis for multi-threaded object-oriented programs. In: PLDI, pp. 115–128 (2003)
39. von Praun, C., Schneider, F.T., Gross, T.R.: Load elimination in the presence of side effects, concurrency and precise exceptions. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 390–405. Springer, Heidelberg (2004)
40. Voung, J.W., Jhala, R., Lerner, S.: Relay: static race detection on millions of lines of code. In: FSE, pp. 205–214 (2007)
41. T.J. Watson Libraries for Analysis (WALA) (2012), http://wala.sourceforge.net/
42. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. In: POPL, pp. 27–40 (2001)
43. Yu, Y., Rodeheffer, T., Chen, W.: RaceTrack: Efficient detection of data race conditions via adaptive tracking. In: SOSP, pp. 221–234 (2005)

Ownership-Based Isolation for Concurrent Actors on Multi-core Machines

Olivier Gruber and Fabienne Boyer

Université de Grenoble
first.last@imag.fr

Abstract. The deep copy of messages that traditionally ensures the memory isolation of actors severely hinders the performance of actor systems on multi-core machines. Several approaches have been proposed in the state of the art to circumvent this overhead, but they require to choose two properties out of the three desired ones: safety, programmability, and efficiency. In this paper, we introduce a novel runtime ownership model that supports the first memory isolation model of actors with these three properties—it is safe, developer-friendly, and efficient.

1 Introduction

In recent years, the actor paradigm has regained much attention as a concurrency model to exploit parallelism in Object-Oriented Programming Languages (OOPL). The argument [7] is that memory isolated actors that cooperate through messaging provides a concurrency model that scales well on multi-core machines. Unfortunately, achieving memory isolation of actors traditionally relies on deep-copying messages—a solution that severely hinders performance [15,4,14,5].

Several approaches have been proposed to circumvent this copy overhead. Some actor frameworks abandon safety for the sake of performance [10,6], advocating to pass messages by reference. These frameworks are efficient but unsafe. Other frameworks [16,7] retain safe messaging and achieve high performance through migrating messages, but at the expense of programmability because of the introduction of special type systems and certain constraints on the shape of permitted messages. A notable exception is the ActorFoundry framework [12] that preserves the programming model of a pass-by-value semantics while avoiding the copy overhead when it is safe to do so. Unfortunately, the proposed static analysis fails to identify all opportunities to optimize out deep copies, producing mixed results from a performance perspective.

This paper discusses a different path that does not require to choose between safety, programmability, and efficiency. With our approach, you can have your cake and eat it too—our solution is safe, developer-friendly, and efficient. Our approach is safe because we guarantee that actors remain strictly memory isolated. It is developer-friendly because messages have unconstrained shapes and we neither introduce any special types nor annotations. Furthermore, developers precisely control if an object belongs to a message or to an actor. To that end,

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 281–301, 2013.

we propose a runtime ownership model based on the concept of first reachability from either a message or an actor. Put simply, an object belongs to a message (resp. actor) if it is first reachable from that message (resp. actor). This ownership model is simple to use and feels completely natural to object-oriented developers. Our solution that migrates messages is efficient compared to passing messages by reference with a typical overhead between 5% to 10%—an overhead that is independent of the shape and size of messages. Furthermore, our approach is multi-core friendly since all our runtime mechanisms are lock-free.

We implemented our complete proposal in the JikesRVM [1], a research high-performance Java virtual machine. We used the actor model of Kilim [16] as a starting point, modifying the actor model as little as necessary in order to integrate our novel memory isolation. We chose Kilim because it is regarded as the best performing framework for Java [9]. All our actor benchmarks ran almost unmodified on both frameworks, the sole meaningful difference being that we require explicit continuations while Kilim proposes automated continuations. Our approach requires modifying the Java virtual machine, adding a write barrier and an extra field in the object header that contains the reference of the owner of an object.

This paper is organized as follows. In Section 2, we quickly recall the main points of the actor paradigm and discuss related work, focusing on memory isolation. In Section 3, we introduce the design of our novel memory isolation based on a runtime ownership model. In Section 4, we illustrate the use of this ownership model through concrete examples. In Section 5 we evaluate our design and we conclude in Section 6.

2 Related Work

The actor paradigm defines an actor system as a collection of concurrent and autonomous entities, called actors, that cooperate through asynchronous messages. Some models propose that messages be sent to actors while other models propose that messages be sent to mailboxes, an actor owning one or more mailbox. In this case, mailboxes are shared objects whose references can be exchanged through messages. The execution model is event-driven, with actors reacting to received messages, one reaction at a time. While each actor is a single-threaded entity, multiple actors execute concurrently, thereby exploiting parallelism. For safe concurrent executions, actors are memory isolated entities, passing messages between actors by value. A straightforward design is to deep-copy messages [15,9,13], which severely degrades the performance of actor systems [9].

For the sake of performance, certain actor frameworks [10,6] abandon safety entirely, advocating a by-reference semantics and explicit deep copies. This approach expects developers to do the right thing, passing messages by reference when it is safe to do so and making deep copies whenever necessary. The approach is attractive for its peak performance and the fact that it preserves standard object-oriented programming practices. The main criticism is undoubtedly the assumption that developers do not make mistakes, which ultimately raises

the question of when can an actor system be trusted to be bug free and therefore safe to use.

The ActorFoundry [12] proposes an original approach that retains the safety of pass-by-value semantics while avoiding the overhead of deep copies when it is safe to do so. To achieve this goal, this framework relies on static analysis that achieves encouraging preliminary results. The analysis combines a novel live variable analysis along with context-sensitive call graph creation and field-sensitive points-to analysis. The approach is especially attractive because it preserves the traditional object-oriented programming model, avoiding the complex and error-prone task of annotating code or using special type systems. Unfortunately, the proposed static analysis still fails to identify all opportunities to optimize out deep copies, producing mixed results from a performance perspective.

Other actor frameworks advocate a pass-by-migration semantics that provides a safe zero-copy messaging. The idea behind migration is that a message can be accessed by only one actor at a time. To achieve this goal, these frameworks control object aliasing through special type systems or annotations [11]. For example, Kilim [16] advocates a static type checking mechanism based on linear types that only allows for tree-shaped messages. Scala [6] has recently proposed a new type system [7] that introduces safe migration. The proposed type systems is interesting because it is somewhat simpler and removes important limitations regarding permitted message shapes. The approach is certainly attractive but the presence of a dual type system raises several questions. First, it is unclear from published papers what are the implications of the restrictions on permitted message shapes. It is unclear if format translations are often necessary, and it they are, what are the performance implications. Second, it is unclear how complex the use of such type systems actually is for the average developer and how severely it impacts traditional object-oriented programming practices.

3 Ownership and Memory Isolation

During our search for an alternative path to provide memory isolation for actor systems, we went through two important design steps, which we retrace in the following two subsections. Both models are based on migration. Since the second step builds on the first, we felt that it helped the clarity of the paper to present the historical evolution of our design.

3.1 Allow Aliasing

The idea behind this first design is quite simple. Rather than trying to prohibit or tightly-control aliasing, like most approaches using special type systems strive to do, we want to allow totally unconstrained aliasing. Our motivation is to maintain object-oriented programming as developers understand it today. If we allow aliasing, we need to shift our focus on preventing the use of illegal references on migrated objects. To better understand illegal references, we should discuss the two different forms of aliasing that create illegal references when migrating

messages: references from the actor state into messages and references from messages to the actor state or into other messages.

The latter form of aliasing can be easily controlled through the use of coloring and a read barrier. We use colors as follows. An actor colors any new object that it creates with its color. As we deep migrate a message, we change the color of each migrated object from the color of its sending actor to the color of its receiving actor. Therefore, a simple read barrier on the color can detect illegal references. In the case of shared objects between two messages, the first migration operation will change the color of the shared objects and the second migration operation will detect an invalid color. In the case of a reference from a message to the actor state, the referenced part of the actor state will be migrated and colored, bringing us back to the former aliasing: the actor keeping references to migrated objects.

This form of aliasing is more difficult to handle properly, even though the principle is straightforward: any reference retained by the sending actor on a migrated object is illegal and must be therefore unusable. The challenge comes from finding these illegal references. Illegal references may be retained in local variables, in arguments of method invocations, and in objects (including arrays). A possible solution could be to scan the actor, garbage-collection style. This would include a complete scan of all objects reachable from both the actor itself and the thread stack of the current reaction. Since the overhead of such a scan would have to be paid at each send; we consider this approach impractical.

We advocate another path: we can allow illegal references to exist as long as we forbid their use. At first glance, the implementation seems rather simple, leveraging a read barrier to check the validity of references before they are used. First, we need a read barrier on reading references out of objects and arrays. Second, we need a read barrier on reading references out of local variables and arguments. But this is not enough since illegal references could also be found on the operand stack[1]. Indeed, any method invocation may send a message and therefore migrate some objects whose references might have been pushed on the operand stack of caller invocations, higher on the thread stack.

At first, this path seems impractical too. First, we can expect an important overhead because of the sheer number of read barriers. Second, the implementation is really delicate because of the necessary scan of the operand stack upon every return of a method invocation. Within an interpreter, one could possibly scan the operand stack after each method invocation, introducing a serious overhead. Within a high performance virtual machine using Just-In-Time (JIT) compilation, the operand stacks is spread across the hardware registers and spill areas in stack frames, complexifying even further the search for illegal references after every method invocation.

Although it seems that we reached a dead end with this design, a simple solution exists if we are to revisit the immediate nature of the send operation. If messages were to be migrated when the current reaction completes rather than

[1] We use Java parlance, focusing on the Java virtual machine for the sake of clarity, although the problem is absolutely not Java specific.

immediately when sent, there would be no need for most of the previous read barriers. In particular, we would not need to bother with any of the read barriers related to the thread stack: local variables, arguments, and operand stack. Since we would be migrating messages once the current reaction completed, the thread stack can be considered as empty as far as memory isolation is concerned. Indeed, when a reaction completes, we are back executing code from the actor framework, and since the framework is the trusted code base, we need not search for illegal references.

Therefore, we propose to introduce the concept of *tail migration*. Developers can continue to shape messages however they please and send them whenever convenient. Each send operation only records the reference of the message, constructing a list of messages that are pending migration. Upon the completion of a reaction, the system automatically processes the pending messages, migrating each one of them to its intended actor. Notice that inter-message aliasing would only be discovered at the time of the effective migration, relying on the above deep-coloring of migrated messages.

Even with tail migration, we must rely on the use of a read barrier on reading references out of objects and arrays. In Java, this means inserting a read barrier on the GETFIELD and AALOAD bytecodes, where GETFIELD loads an object reference from an object field and AALOAD loads an object reference from an array object. With this read barrier in place, illegal references can be retained by objects and arrays, but they cannot be used, so safety is guaranteed.

Even though safety is guaranteed, we must consider the following point. Since an actor can retain illegal references, it can potentially force large graphs of objects to stay alive in other actors. This suggests to extend the garbage collector so that it discovers illegal references and nullifies them. While nullifying illegal references solves the problem, it does not seem an appropriate solution from a programming perspective since it silently removes any trace of memory isolation violations, thereby concealing the illegal behaviors of certain actors. We propose to use a special bit pattern instead of null; this special bit pattern would be treated by the garbage collector as a null reference, but it would be treated by the read barrier as an illegal reference, raising an appropriate exception such as IllegalPointerException.

To summarize, this design preserves the traditional programming style of OOPLs and provides the safe migration of unconstrained graphs of objects. Unfortunately, we can expect a relatively high overhead since it relies on a read barrier [2,17]. Furthermore, we feel that the lazy discovery of illegal references induces a programming model that is just too cumbersome for developers. Indeed, cross-message references are only discovered when the reaction completes, not when they are created. Even worse, other illegal references are only discovered if there is an attempt to use them, which means that the threat of illegal pointer exceptions always remains. Furthermore, it is our experience that understanding the real source of these exceptions when they finally occur is a daunting task for most developers, therefore greatly limiting the usability of this approach in practice.

3.2 Control Aliasing

Our second design builds on the first, combining tail migration with a novel ownership model. Our ownership model defines two *owner classes*: the Actor class and the Message class. In other words, each actor and each message are individual owners. Each owner defines one ownership domain including the owner object and all the objects owned by that owner. Therefore, owner objects are created as owning themselves. Other objects are created free (not owned) and will remain free until garbage collected or until absorbed by an ownership domain. A free object is absorbed as soon as it becomes reachable from an owner object, directly or indirectly. In other words, the ownership propagates through reference assignments:

$$L.f = R; \tag{1}$$

When the field f of the left-hand-side object L, owned by an owner O, refers to a right-hand-side object R, the ownership propagates from L to R as follows:

1. **If the object L is free**, there is no ownership to propagate. If the object R was free, it remains free. If it was owned, its owner remains unchanged.
2. **Object L is owned and R is free**. When a free object is first referenced by an owned object, we propagate the ownership: R becomes owned by the owner of L.
3. **The objects L and R are owned**. The assignment is illegal unless L and R are owned by the same owner.
4. **The object L is owned and R is shared**. The assignment is always legal and there is no ownership propagation.

Even with isolated ownership domains, the concept of shared objects is necessary to model shared references to trusted language objects. For instance, mailbox objects are typically shared by actors to send and receive messages. Other language objects must also be shared such as Java enumeration, Java classes, or class loader strings. In general, it is also accepted that immutable objects ought to be shared. Shared objects are never absorbed, they remain free until garbage collected. However, it is important to assert that, aside from references to well-identified shared objects, ownership domains are entirely isolated. Any assignment attempting to create a reference across the boundary between two ownership domains would raise an *illegal assignment exception*, immediately identifying the source of a future illegal reference.

It is important to point out that the absorption of an object is a *deep absorption*, applying the same write barrier on all encountered references. Indeed, in the last case above, despite the fact that R is free, the graph of objects reachable from R might be composed of either owned or free objects. Free objects are absorbed but objects owned by other owners would represent an illegal situation forcing an exception to be thrown. It is also important to point out that once

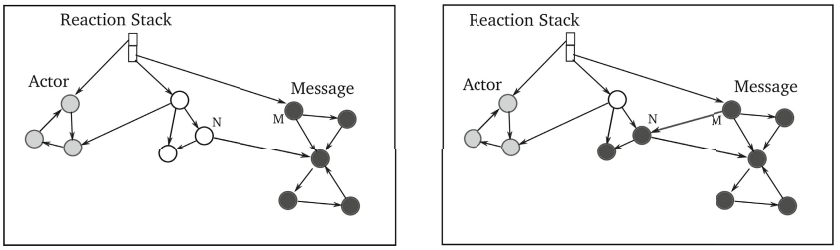


Fig. 1. Ownership Propagation

owned, an object remains owned by the same owner until it becomes garbage and it is reclaimed.

Figure 1 illustrates an example of ownership propagation. On the left-hand side, we have an actor, the stack of its current reaction, and a message M. There is also a small graph of free objects, only reachable from the reaction stack. Going to the right-hand side, the actor is adding the object N to the message M, which propagates the ownership of the message M onto the object N and those reachable from N.

Note that propagating ownership does not entail any object copy. It is similar to coloring but instead of propagating a color, we propagate the identity of the owner object. In Java, this means the above write barrier must be added to the PUTFIELD and AASTORE bytecodes, PUTFIELD stores an object reference in an object field and AASTORE stores an object reference in an array object. The simplest implementation is to have an extra hidden reference per object, called the *owner reference*. When the object is free, that owner reference is null. When the object is owned, that owner reference refers to the owner object.

Notice that local variables, method arguments, and free objects are allowed to refer to any object—free or owned. This is important because this allows a total programming freedom when manipulating the state of an actor or of a message. In other words, despite the fact that objects may be owned, either by the actor or by a message, the actor developers retain a complete programming freedom. For instance, an enumeration on a hash table would be a free object, even if the hash table is owned. The enumeration object will be using direct references on owned objects that are internal to the collection implementation. In other words, aliasing of owned objects through free objects is legal and not limited in any way. This model is correct, never endangering isolation, since we adopted a tail migration rather than an immediate migration when sending messages.

4 Examples

In this section, we illustrate the use of our ownership model through a simple yet complete example. The examples are written in Ownership-Kilim (O-Kilim), our actor framework based on Kilim [16]. The purpose of this section is two-fold. First, we want to illustrate the minimal differences between O-Kilim and

Kilim. Second, we want to highlight how natural our ownership model feels to object-oriented developers.

Like Kilim, we have three special classes: the Task class, the Mailbox class, and the Message class. The Task class is the concept of actor in Kilim. The three classes are part of the trusted code base, but only the Mailbox class is final. Mailboxes are shared across tasks, allowing tasks to send and receive messages. There are two key differences between Kilim and our proposal. First, O-Kilim relies on our ownership model. Second, Kilim advocates automated continuations, relying on blocking operations on mailboxes with an automated wind-unwind of the thread stack upon blocking and resuming tasks. O-Kilim assumes explicit continuations and run-to-completion reactions. The O-Kilim framework guarantees that each actor executes at most one reaction at any one time, irrespective of the number of mailboxes it is bound to. An actor binds to a mailbox by registering a listener, whose MailboxListener interface is given in Listing 1.1. Only one actor may bind to a given mailbox.

Developers are expected to extend the Actor class and Message class as they see fit. Instances of both the Actor and Message class are owners, each instance defining a separate ownership domain. Remember that owners are instantiated as owning themselves and therefore their constructors already execute within the confinement of their ownership domain. In Listing 1.1, we also show the simplest actor constructor that first binds to a given mailbox and then starts.

Listing 1.1. Server Actor

```
1  public interface MailboxListener<T> {
2    void onMessage(Mailbox<Message<T>> mb, Message<T> msg);
3  }
4
5  public class Server extends Task implements MailboxListener<Message> {
6    Mailbox<Message> mymb;
7    public Server(Mailbox<Message> mb) {
8      mymb = mb;
9      mymb.addMessageSubscriber(this);
10     start();
11   }
12   public void onMessage(Mailbox<Message> mb, Message msg) {
13     Mailbox<Message> niu = new Mailbox<Message>();
14     new Parser(niu);
15     niu.put(msg);
16     return;
17   }
18 }
```

In Listing 1.1, the server's reaction creates actors (Parser) that concurrently process received requests. Each request will be about parsing a text document into an in-memory W^3C document. Notice that messages can be forwarded

without hassle, one of the main performance advantages of migration. Line 15, we can see that forwarding a message is as simple as putting into a mailbox, which does not migrate the message immediately but remembers that it should. At the end of the reaction, line 16, the message will be migrated, without any overhead.

Notice also that mailboxes and actors are created as regular objects. Notice however that created actors are not referenced; in fact, actors are not shared objects and cannot be referenced across ownership domains. Furthermore, the reference to an actor is only legal within local variables and arguments belonging to a reaction on that actor. Hence line 14, the newly created actors cannot be referenced; indeed, any communication with an actor must happen through mailboxes.

Listing 1.2. Main Initialization

```
1  public static void main(String args[]) {
2    Mailbox<Message> mbserver = new Mailbox<Message>();
3    new Server(mbserver);
4    int maxClient = Integer.parse(args[0]);
5    for (int i = 0; i < maxClients; i++) {
6      Mailbox<Message> niu = new Mailbox<Message>();
7      new Client(i, niu, mbserver);
8      niu.put(new Message(Message.START, args[i+1]));
9    }
10 }
```

Listing 1.2 illustrates the corresponding main initialization of an actor system. We create one server actor and a certain number of client actors, giving to each client the mailbox of the server and a url of a document. Listing 1.3 shows the simple message class we use throughout this example.

Listing 1.3. Message Class

```
public class Message extends kilim.Message {
  public static final int START = 1;
  public static final int REQUEST = 2;
  public static final int RESULT = 3;
  int kind;
  Mailbox mb;
  String url;
  Document doc;
  Message(int kind, String url) { ... }
  Message(int kind, Mailbox mb, byte[] text) { ... }
  Message(int kind, Document doc) { ... }
}
```

Listing 1.4 shows the client actor, reading the text document from the url (line 17), requesting the server to parse the document (line 18) and absorbing the in-memory XML document when receiving it (lines 22 to 24). Notice the extraction (line 23) before the absorption (line 24). Indeed, the XML document is owned by the message, it must therefore be extracted before it can be absorbed by the actor. Also notice the use of a local variable to remember the document reference since the msg.doc field will be nullified by the extraction, preserving the invariants of our ownership model.

Listing 1.4. Client Actor

```
1  public class Client extends Task implements MailboxListener<Message> {
2    Mailbox<Message> mymb;
3    Mailbox<Message> mbserver;
4    int id;
5    Document doc;
6    public Client(int no, Mailbox<Message> mb, Mailbox<Message> mbs) {
7      id = no;
8      mbserver = mbs;
9      mymb = mb;
10     mymb.addMessageSubscriber(this);
11     start();
12   }
13   public void onMessage(Mailbox<Message> mb, Message msg) {
14     byte[] text;
15     switch (msg.type) {
16     case Message.START:
17       text = readDocument(msg.url); // read text document
18       mbserver.put(new Message(Message.REQUEST, mymb, text));
19       break;
20     case Message.RESULT:
21       Document tmp = msg.doc
22       if (filter(tmp)) {
23         extract(tmp);
24         doc = tmp;
25       }
26       break;
27     }
28   }
29 }
```

The extraction of an object from an ownership domain is a deep extraction, recursively forcing all reachable objects to be free again. Also, the extraction of an object from its ownership domain nullifies any remaining reference within that ownership domain that refers to objects that were just freed. This is important to preserve our invariants: (i) there is no references from an ownership domain to free objects, (ii)ownership domains remain fully encapsulated: there cannot

be any cross-domain references stored in reference fields of objects (including arrays).

However, notice (line 21 and 22) that messages can be freely manipulated from local variables and arguments, including references on internal objects. We believe this is one major advantage of our approach—one that preserves a completely natural programming model for object-oriented developers. The method filter (called line 22) would scan the document and decide if it should be absorbed or not. This filter method would be written exactly as it would be written in pure Java. The only constraint in our model is that cross-domain references, between two messages or between a message and an actor, are illegal. However, one can always extract an object from one domain and absorb it into another. This is exactly what happens line 23 that extracts the document from the ownership domain of the message and line 24 that absorbs the document in the actor state.

And there is nothing more complex than that in our model in order to control ownership and therefore ensure the safe isolation of actors. The rule could not be simpler: just assign the reference of your object were it belongs first, then alias it through local variables and arguments as necessary to manipulate it. It is our experience that this model feels natural to object-oriented developers; they write code as they are used to.

5 Evaluation

To validate our novel design for software memory isolation based on the safe migration of unconstrained messages, we implemented O-Kilim on top of a modified JikesRVM—a high performance research virtual machine for Java. We modified the virtual machine to include our ownership and we implemented the O-Kilim framework on top, providing tasks (actors) and mailboxes with a tail-migration semantics. The execution model is event-based, one actor reacting to only one message at a time, with each reaction running to completion. However, our framework may use multiple worker threads to execute multiple reactions concurrently across multiple actors.

We modified the JikesRVM version 3.1.0, implementing our write barrier in both the baseline compiler and the optimized compiler. All given numbers are obtained with the optimized compiler, with the O2 optimization level for both the boot image and the benchmark code. The JikesRVM is setup so that it does not use dynamic recompilation at runtime. In other words, Java code is compiled only once and runtime statistics are turned off. All benchmarks are run with a warmup run that forces all Java code to be compiled at the O2 level. Then, we measure ten successive runs, during which there is no longer any compilation overhead since we turned off dynamic recompilation. The given numbers are the mean average of the ten timed runs. We used the Immix garbage collection, with a maximum heap size of 2GB.

All experiments were run on an HP-Z400, with an Intel(R) Xeon(R) CPU W3520@2.67GHz, 64bit, 4 cores, 8 threads, 8GB RAM with memory bus at

2.4GHz, 4x32KB L1 for both data and instructions, and 4x256KB L2, and 8MB L3. Ubuntu 12.04 is installed, 64bit version, with the Linux kernel version 2.6.35.

5.1 Actor Benchmarks

Although there is no official benchmark for actor frameworks [9], three typical benchmarks have been consistently used in the literature [9,16,8].

1. The *Ring benchmark* evaluates message passing, a ring of actors are passing a token message around the ring a certain number of times. Our default setting is to pass 4 million times the token around 500 actors.
2. The *Chameneos benchmark* [8] has N creatures, called Chameneos, that have a cyclic behavior where they request a broker to arrange a meeting between two creatures and when the meeting takes place, the two creatures play together for a while and change color. Our default setting is to have 20 creatures, meeting one million times. To simulate playing, we have creatures change color a certain number of times at each meeting.
3. The *QuickSort benchmark* illustrates a concurrent sorting service based on a client-server pattern. The service is implemented by an actor that receives multiple requests from client actors, each one to sort one collection. Concurrent requests are processed in parallel. For each request, the service creates an actor and forwards the collection to sort. The created actor sorts the collection and sends back the sorted collection to the client. Each client has a cyclic behavior: creates a collection, requests that the collection be sorted, waits for the sorted collection, and then scans the sorted collections to check that it is indeed sorted. We use 500 clients that each issue 100 requests to sort a collection with 100 elements. The collection is a Java list of comparable objects.

With these benchmarks, we can actually establish that our framework has a performance behavior that is comparable to Kilim, despite the slight changes in the actor model: run-to-completion reactions and the presence of our ownership model. Having established the soundness of our prototype with respect to one of top-performing actor frameworks [9], we can feel confident that our comparison of different memory isolation schemes is meaningful.

First of all, it is important to assess how similar the benchmarks are when running them on O-Kilim versus Kilim. The benchmarks are not only algorithmically the same, but they are almost identical syntactically since we have preserved the Kilim concepts of Tasks (actors) and Mailboxes, almost unmodified. Rather than offering a blocking operation to get messages from a mailbox, we offer a callback mechanism to notify an actor that a message is available, triggering the actor's reaction[2]. This is actually the only syntactical difference as our ownership is transparent, as illustrated in our example in Section 4.

[2] For all benchmarks, explicit continuations were trivial, something that might not always be true for all applications.

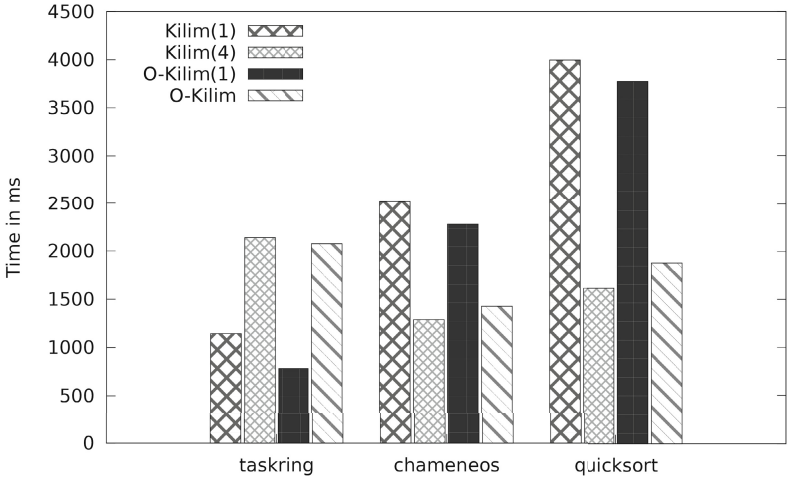


Fig. 2. Unsafe O-Kilim vs Unsafe Kilim

In Figure 2, we ran our benchmarks on Kilim (version 0.6) and on O-Kilim, both sending messages by reference. In other words, we measure the Java performance of both frameworks, providing no memory isolation in this first experiment. We ran all benchmarks with either one or four worker threads, a worker thread being a Java thread solely used to run actors' reactions. At first glance, we can observe that both frameworks behave remarkably the same across all benchmarks.

Regarding the Ring benchmark, both frameworks are unable to benefit from multiple workers as the benchmark has no builtin parallelism. Unsurprisingly, it is more efficient to schedule a sequence of reactions on a single thread than passing them around multiple threads, experiencing the delays necessary to wake up worker threads that are sleeping. However, both frameworks are able to benefit from using multiple worker threads on the two other benchmarks (the Chameneos and QuickSort). We can notice that single-threaded performance are close across all benchmarks and so are the achieved speedups with four workers. From these performance figures, we deduce that our O-Kilim prototype is a fair vehicle, compared to Kilim, in order to evaluate different memory isolation schemes.

Focusing on memory isolation, we compare different message passing schemes in Figure 3. The benchmarks are the same as before, but we run them on a single-threaded version of the O-Kilim framework. The rationale is that the overhead of memory isolation is a single-threaded overhead since it incurs no locking and no concurrent behavior. The *by-ref scheme* means that we ran the benchmarks on the unmodified JikesRVM, running the O-Kilim framework that passes messages by Java reference. This is equivalent to what we showed in Figure 2 under O-Kilim numbers. The *migration scheme* means that we ran the benchmarks on a modified JikesRVM to include our ownership model, running the O-Kilim framework. We

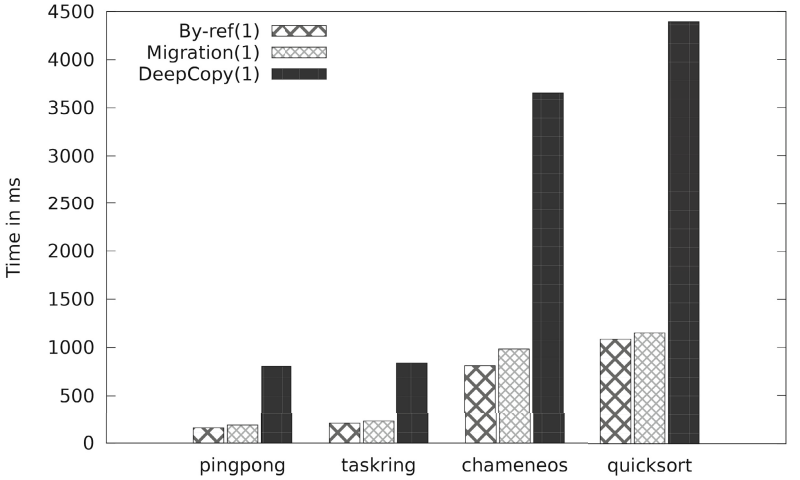


Fig. 3. Messaging Overheads

discuss the details of the corresponding modifications of the JikesRVM in the next subsection. The *deep-copy scheme* means that we ran the benchmarks on a modified JikesRVM in which we added an optimized deep-copy mechanism, but there is no write barrier of course. To implement this deep copy, we use the low-level capabilities of the JikesRVM to efficiently scan and clone objects, combined with the use of a hash table to preserve cycles when deep copying.

These performance figures confirm the established belief [9,12,7] that the deep-copy scheme severely hinders performance. This is even the case despite the fact that we rely on an optimized deep copy and not the costly Java serialization [13]. The Table 1 summarizes the overheads of the Figure 3. All measurements are done with the O-Kilim framework and the same benchmarking code, only modifying the semantics of the send operation in the O-Kilim framework.

Table 1. Deep-copy vs Migration Overheads

-	**Ring**	**Chameneos**	**QuickSort**
By-ref	783ms	2279ms	2505ms
Deep-Copy	3190ms	5072ms	8568ms
overhead	247%	122%	265%
Migration	917ms	2491ms	2761ms
overhead	17.11%	9.3%	10.21%

The Ring benchmark measures a pure messaging overhead since each actor is creating a message that it passes to the next actor down the ring. Consequently, this benchmark gives us an estimate of the overhead of creating and

sending small messages—a single object containing a single integer field (primitive type). The deep-copy overhead is high, even though the messages are small. The reason is that there is no processing at all to balance any messaging overhead. The same is true for migrating message, so the Ring benchmark measures our worst overhead, which is around 20%. The Chameneos benchmark shows the positive effect of adding processing. Both the deep-copy and migration benefits from added processing, with overheads that are cut in half. However, the QuickSort benchmark shows that the deep-copy overhead soars with larger messages whereas the migration overhead remains identical, around 10%.

From these figures, we confirm previously published results [15,4,14,5] stating that a deep-copy approach is much more costly than migrating messages. We focus now on characterizing more accurately our overhead, changing the amount of processing as well as the shape and size of the processed data. First, we will change the amount of processing in the Chameneos benchmark, varying the number of times meeting creatures are changing colors.

Table 2. Varying Processing in Chameneos Benchmark

Color Changes	**1**	**50**	**100**
Time(1W)	985ms	2491ms	3940ms
overhead	21.45%	9.30%	4.45%
Time(4W)	1345ms	1583ms	1875ms
overhead	14.37%	10.85%	0.4%
ratio	1.36	0.63	0.47

The performance figures are in Table 2. The *Time(1W)* line gives the execution times for the Chameneos benchmark, on a single worker, with 20 creatures, changing colors at each meeting either once, fifty times, or one hundred times. Changing color is a simple method with two nested switch with 3 cases each, deciding the new color of a creature based on its current color and the color the other creature had when the meeting began. The added processing does not entail any messaging, it is just a pure Java method manipulating Java enums. The *Time(4W)* line gives the execution times for the same Chameneos benchmark but using four worker threads.

Notice our high overhead when creatures only change color once (very limited processing per message). We are back to an overhead of around 20%, as it was the case with the Ring benchmark earlier. Also notice that when processing per message is limited, our ability to leverage multiple workers is challenged—we actually execute 1.36 slower with four workers than with one. In contrast, notice the positive effect of adding processing, with decreasing overheads for 50 and 100 color change. Despite the fact that the added processing is really small our overhead drops significantly: adding 1.4 micro-seconds per meeting drops our overhead to 9.30% and adding 2.8 micro-seconds per meeting drops it to 4.45%.

Moreover, notice the positive effects on our ability to leverage multiple workers. With 50 color changes, we reduce the execution time by 0.63 with an overhead at 10.85%. With 100 color changes, we reduce the execution time by 0.47, with an overhead at 0.4%. We believe that these figures indicate that modern processors are able to absorb our isolation overhead through concurrent execution (both hyper-threaded cores and advanced superscalar architectures) when there is enough processing per message. An effect that we will be confirmed on a novel XML benchmark below.

But before moving to XML processing, we can question the validity of our remarks since we only added synthetic processing (a loop manipulating Java enums). Consequently, we also conducted similar experiments with the QuickSort benchmark, varying not only the shape and size of the sorted collections, but also the cost of the element comparison.

Table 3. Varying Processing in QuickSort Benchmark

Settings	**A**	**B**	**C**
Time(1W)	2761ms	4972ms	10609ms
overhead	10.21%	9.05%	8.66%
Time(4W)	1234ms	2250ms	3998ms
overhead	7.39%	3.11%	4.14%
ratio	0.44	0.45	0.37

The performance figures are in Table 3. All the three settings are on collections of one hundred elements, using the unmodified QuickSort benchmark. In the first setting, called A, each collection is a Java ArrayList and elements are a simple object, with one integer field. The comparison of elements is therefore cheap: comparing primitive integers. In the second setting, called B, the elements are still simple objects, but they contain a string rather than an integer. In fact, the string is the textual representation of the integer value they contained in the setting A. The goal is to increase the cost of comparing elements, now comparing strings rather than primitive integers. In the third setting, we change the implementation of the collections, moving from Java ArrayList to Java LinkedList. The goal is to increase the overhead of accessing the elements of the sorted collection during the sort. The performance figures in Table 3 confirm our previous analysis. Across all sizes and shapes, our single-threaded overhead is around 10%. Furthermore, this overhead drops to 3% with parallel execution on four workers. Additionally, we maintain our speedup, executing up to 0.37 times faster on four workers—a speedup of 2.7 on 4 workers.

To confirm these statements, we conducted one last experiment with an *XML Parser benchmark*. We modified our QuickSort benchmark to parse XML documents concurrently rather than sorting Java collections. In other words, we retained the overall architecture of the benchmark, but we replaced our homegrown QuickSort algorithm with the use of the Java SAX parser. We use 100 clients that each issue 50 requests to parse a XML text document into a W^3C

document. The text document is a simple array of characters, while the W^3C document is a complex in-memory graph of objects, whose exact size and shape depends on the parser implementation.

Table 4. Varying Processing in XML Parser Benchmark

Settings	**Small**	**Medium**	**Large**
Time(1W)	1251ms	5664ms	17102ms
overhead	5.2%	7.68%	7.66%
Time(4W)	604ms	2926ms	7975ms
overhead	0.16%	9.13%	0%
ratio	0.48	0.51	0.46

The performance figures are in Table 4. Again, the performance figures confirm our analysis, even with a completely different processing and different data sets. The small XML document is really small with 7 tags, 5 attributes, accounting for 161 bytes. The medium document is more reasonable, with 69 tags and 32 attributes, accounting for 1217 bytes. The large document has 225 tags and 104 attributes, accounting for 5357 bytes. In this last experiment, with realistic processing, our overhead is around 5-8% with single-threaded execution and drops to zero overhead with four workers, while retaining an acceptable speedup, executing at twice the speed with 4 workers compared to one worker.

Overall, we believe that we have established that our memory isolation scheme induces a low overhead and does not impede concurrent execution. We further believe that we have reached our goal since we achieve low-cost memory isolation with minimal changes to the actor model and no change at all to the habits of object-oriented developers. However, we feel that more work is necessary on two fronts. First, a comparison analysis with solutions based on static analysis, that also preserve the object-oriented programming model, seems necessary. Second, the actor community must establish a representative benchmark suite for multi-core machines, with realistic concurrent applications that are representative of the concurrent and processing patterns targeted by actor systems.

5.2 WriteBarrier Benchmarking

In this section, we detail the design of our memory isolation mechanisms. In particular, we detail our ownership implementation, the associated write barrier. We explain how we integrated these mechanisms in the JikesRVM virtual machine. We also provide low-level performance numbers from the various benchmarks we ran in the previous subsection, allowing us to further explain the excellent performance figures we discussed earlier.

Our ownership model has a straightforward implementation. We added an *owner field* to the header of each Java object. This owner field holds the reference to the owner of the object, or null otherwise. This reference is known to the garbage collector, so a live object maintains its owner alive. The added overhead

to the garbage collection process is negligible. As explained earlier, the owner field is assigned by a deep absorption triggered by our write barrier.

The concept of write barrier is not new; it has been used intensively for supporting various garbage collection schemes [17]. In fact, the Immix garbage collector [3] we use in all our experiments uses write barriers for its own tracking of references. Traditionally, the reported performance overhead of write barriers [2,17] is typically around 2% to 6%. Traditionally, a write barrier is divided into a fast path and a slow path. The fast path handles the common case, requiring very few instructions; instructions that are inlined by the Just-In-Time compiler (JIT) on compiling the PUTFIELD and AASTORE bytecodes. The slow path handles the exceptional cases, requiring more instructions that are not inlined, therefore requiring that the fast path branches to the slow path when an exceptional condition occurs.

Listing 1.5. Write Barrier

```
static void writeBarrier(Object left, Object right) {
  if (right!=null) {
    Object lowner, rowner; // left and right owners.
    lowner = Magic.getObjectAtOffset(left,JavaHeader.OWNER_OFFSET);
    if (lowner!=null) {
      rowner = Magic.getObjectAtOffset(right,JavaHeader.OWNER_OFFSET);
      if (lowner!=rowner)
        writeBarrierSlowPath(lowner,left,rowner,right);
    }
  }
}
```

```
Assembly:
   ; ECX = right
   ; EDX = left
   TEST  ECX,ECX      ;
   JEQ                ; branch if right==null
   MOV EAX  -24[EDX]  ; EAX = lowner
   TEST EAX,EAX       ;
   JEQ                ; branch if lowner==null
   MOV EBX, -24[ECX]  ; EBX = rowner
   CMP EAX            ; compare lowner and owner
   JNE                ; branch to slow path if lowner!=rowner
```

Our write barrier is no exception to this split into a fast path and a slow path. Our fast path is given in Listing 1.5, both in Java and the corresponding IA-32 assembly code. The write barrier is written in Java because the JikesRVM is a meta-circular Java Virtual Machine (JVM), written itself in Java. We insert our write barriers like the garbage collector does, when the JIT compiler expands runtime services, after the final High-Level Intermediate Representation (HIR) is produced and before it is lowered to the Low-level Intermediate Representation (LIR). We have not touched the LIR translation to machine code. Since our

write barrier is inserted on assignment bytecodes (PUTFIELD and AASTORE), we have a left-hand-side object and a right-hand-side object, called left and right in the Java source. Notice the added owner field in the object header, holding the reference to the owner if any.

Our fast path captures the most common cases: (i) when free objects are assigned (left owner is null) and (ii) when legal references are assigned within an ownership domain (left and right owners are the same). The Table 5 gives the number of write barriers and the decomposition into fast and slow paths for various benchmarks, reusing previous settings. For Chameneos, we reused 20 creatures, 1000000 rendez-vous, and 50 color change per meeting. For the XML parser, we reused 100 actors, 50 requests over our medium-size document. For QuickSort, we reused 500 clients, 100 requests on a array list of 50 objects.

Table 5. Write Barrier and Absorption

	Ring	Chameneos	QuickSort	Parser
Messages	4M	4M	50,000	10,000
WriteBarrier	4M	16M	40M	21.4M
fast	1%	35%	32%	99%
slow	99%	65%	68%	1%
Absorption	4M	4M	10.2M	4.4M
Extraction	0	0	5.1M	2.2M
Overhead	17.11%	9.3%	10.21%	7.68%
total exec time	917ms	2780ms	6658ms	1488ms

Table 5 also shows other performance numbers related to the absorption or extraction of objects. We give the number of absorbed and extracted objects. We also recall the total execution times and the overhead induced by memory isolation. These numbers show that our various benchmarks do cover a wide mix of overheads. We cover from 4 million write barriers up to 40 millions, with different fast/slow ratios, from 1% up to 99% fast paths. The number of absorbed objects also varies greatly, from 1 million up to 10 millions, and so is the number of extracted objects, from 0 to 5 millions. This variety makes us believe that our performance figures given earlier are indeed representative of the quality of our proposed design for the safe memory isolation of actors.

6 Conclusion

This paper proposed a novel runtime ownership model that provides object-oriented developers with a completely natural programming model. Our ownership model does not require special types or annotations; the model is straightforward, an object belongs to the first owner it is reachable from. This rule is easy to understand and simple to put into practice in object-oriented

languages. Regarding the actor model it is integrated in, our approach has essentially one requirement: to use an event-driven execution that permits the tail-migration of messages. Our approach does not constrain the shape of permitted messages, any shape or size can be migrated at no cost. The overhead of our approach comes from the ownership management (write barrier, absorption, and extraction). This overhead is less than 20% across all our benchmarks and can typically be expected to be around 5%. This work suggests several directions for future work. First, it would be interesting to see how different static analysis techniques could help reduce the overhead of our ownership model. This is a research direction that we feel promising and that we intend to pursue. Second, we feel that it would be important for the actor community to develop realistic benchmarks that would help stronger evaluations of different actor systems.

Acknowledgments. We wish to thank Laurent Daynès for his comments on this work and valuable insights on Isolates and XIMI in particular. We also thanks the anonymous reviewers for their valuable comments.

References

1. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V., Trapp, M.: The Jikes Research Virtual Machine project: Building an open source research community. IBM Systems Journal 44(2) (May 2005)
2. Blackburn, S.M., Hosking, A.L.: Barriers: friend or foe? In: ISMM 2004: Proceedings of the 4th International Symposium on Memory Management, pp. 143–151. ACM, New York (2004)
3. Blackburn, S.M., McKinley, K.S.: Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (June 2008)
4. Czajkowski, G.: Application isolation in the Java Virtual Machine. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 354–366 (October 2000)
5. Golm, M., Kleinöder, J., Bellosa, F.: Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Operating Systems (HotOS), Elmau/Oberbayern, Germany, pp. 3–8 (May 2001)
6. Haller, P., Odersky, M.: Actors that unify threads and events. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 171–190. Springer, Heidelberg (2007)
7. 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)
8. Kaiser, C., Pradat-Peyre, J.-F.: Chameneos, a concurrency game for java, ada and others. In: ACS/IEEE International Conference on Computer Systems and Applications. Book of Abstracts, p. 62 (January 2003)
9. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: a comparative analysis. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, pp. 11–20. ACM, New York (2009)

10. Rettig, M.: JetLang (2008-2009), http://code.google.com/p/jetlang
11. Müller, P., Rudich, A.: Ownership transfer in universe types. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA 2007, pp. 461–478. ACM, New York (2007)
12. Negara, S., Karmani, R.K., Agha, G.: Inferring ownership transfer for efficient message passing. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 81–90. ACM, New York (2011)
13. Palacz, K., Czajkowski, G., Daynes, L., Vitek, J.: Incommunicado: Efficient Communication for Isolates. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 262–274 (November 2002)
14. Java Community Process. Application Isolation API Specification, http://jcp.org/en/jsr/detail?id=121
15. 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)
16. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
17. Yang, X., Blackburn, S.M.B., Frampton, D., Hosking, A.L.: Barriers reconsidered, friendlier still! In: Proceedings of the Eleventh ACM SIGPLAN International Symposium on Memory Management, ISMM 2012, Beijing, China, June 15-16 (2012)

Why Do Scala Developers Mix the Actor Model with other Concurrency Models?

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson

Department of Computer Science, University of Illinois at Urbana–Champaign, USA
{tasharo1,rjohnson}@illinois.edu, pdinges@acm.org

Abstract. Mixing the actor model with other concurrency models in a single program can break the actor abstraction. This increases the chance of creating deadlocks and data races—two mistakes that are hard to make with actors. Furthermore, it prevents the use of many advanced testing, modeling, and verification tools for actors, as these require *pure* actor programs. This study is the first to point out the phenomenon of mixing concurrency models by Scala developers and to systematically identify the factors leading to it. We studied 15 large, mature, and actively maintained actor programs written in Scala and found that 80% of them mix the actor model with another concurrency model. Consequently, a large part of real-world actor programs does not use actors to their fullest advantage. Inspection of the programs and discussion with the developers reveal two reasons for mixing that can be influenced by researchers and library-builders: weaknesses in the actor library implementations, and shortcomings of the actor model itself.

1 Introduction

The actor model [1] for concurrent and parallel programming is gaining popularity as multi-core architectures and computing clouds become common platforms. The model's restriction of communication to asynchronous message-passing simplifies reasoning about concurrency, guarantees scalability, allows distributing the program over the network, and enables efficient tools for testing [17,32], modeling [30] and verifying [33] actor programs.

These advantages, however, depend on an intact actor abstraction. Programmers mixing the actor model with other concurrency models can easily break the abstraction. It increases their chance of committing mistakes that the actor semantics carefully avoid: shared state between actors breaks transparent distribution and can introduce fine-grained data races; and blocking and synchronous communication can lead to deadlocks. Furthermore, most of the tools for testing actor programs lose their bug detection capability and their efficiency when used on programs that mix concurrency models.

When examining Scala [23] programs available from public github[1] repositories that use either the original Scala actor library [11] or Akka [5], we discovered

[1] https://github.com

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 302–326, 2013.

that many of the programs mix actors with other kinds of concurrent entities such as Java threads.

This raised the question why programmers gave up the advantages of actors and mixed them with threads. Was it a temporary measure, as the programmers converted thread-based parallelism to actors? Does this indicate problems with the actor model, with the implementation of the actor libraries for Scala, or in the education of Scala programmers?

In this paper, we formulate three research questions to study the phenomenon of mixing concurrency models:

RQ1. *How often do Scala programmers mix actors with other kinds of concurrent entities?* This question obviously goes far beyond Scala, but we decided to look first at Scala before looking at other languages.

RQ2. *How many of the programs are distributed over the network, and does distribution influence the way programmers mix concurrency models?* Our motivation for this question is that the actor model can be used to exploit multiple local processors, as well as to distribute the program over the network. Hence, one reason for mixing concurrency models could be that some models are better for particular kinds of programming than others.

RQ3. *How often do the actors in the programs use communication mechanisms other than asynchronous messaging?* Communication through asynchronous messaging reduces the possibility of deadlock and data races, which are common problems in the shared-memory model. However, in Scala, actors can also communicate via other mechanisms such as shared locks. The motivation for this research question is to find out if mixing the actor model with other concurrency models is related to the advantages of asynchronous communication, that is, whether developers use actors for those parts of the program that have high risk of data races or deadlocks.

This paper describes how we selected programs to study (Section 3), the way we measured them, the resulting measurements (Section 4), and the conclusions we drew. We also contacted the developers, and they provided many insights into the meaning of our observations (Section 5). Our findings (Section 6) reveal that the reasons for mixing the actor model with other concurrency models are mostly due to weaknesses in the implementations of the libraries. However, they also show weaknesses in the actor model itself, as well as in the experience of developers.

In summary, this work makes the following contributions:

1. It is the first to point out the phenomenon of Scala developers mixing the actor model with other concurrency models. This phenomenon is at odds with the accepted wisdom about the actor model, which says that its benefits from no shared state and asynchronous communication outweigh its drawbacks.
2. It gives statistics about mixing actors with other kinds of concurrent entities in real-world Scala programs.
3. It gives recommendations for researchers and actor-library designers.

2 Background: Concurrent Programming with Actors

Actors [1,13] are a model of concurrent programming. They are concurrently executing objects that communicate exclusively via asynchronous messages. Each actor buffers the messages it receives in a mailbox and processes them sequentially, one at a time. Upon processing a message, an actor can change its state, send messages, or create other actors. The event-based computation model avoids blocking waits for specific messages, which helps to keep clear of deadlocks in the system. Each message is processed in an atomic step [2]. This reduces nondeterminism in actor programs and makes reasoning about the program easier.

Actor semantics furthermore mandate that each actor is perfectly encapsulated, that is, there is no shared state between actors. This greatly reduces the potential for data races. In combination with asynchronous execution, the lack of shared state allows actor programs to fully exploit the processing cores of current—and future—multi-core processors. Hence, actors offer strong local scalability, which makes them an attractive programming model for modern architectures.

Another trait of actor semantics is location transparent addressing: actors know each other only by unique, opaque addresses. Not having to specify the (physical) location of a message recipient allows the run-time system to distribute the actors that constitute a program across a computing cluster. Consequently, the actor model provides scalability beyond single machines.

Actor Libraries. Obtaining the scalability benefits does not require a language that enforces *strict* actor semantics; it is sufficient to have a library providing asynchronous messaging between concurrent objects, and to adhere to coding conventions for avoiding shared state. This allows programmers to reap the scalability-benefits of the actor model, and to break the abstraction if desired—for example by introducing shared state between actors or using non-actor concurrency constructs.

Scala [23]—an object–functional language that runs on the Java virtual machine—is one of the most popular languages that follow this path. Its standard library provides non-strict actors as the default concurrency mechanism. We refer to the actors of this implementation as *Scala actors* [11]. Building upon experience from Scala actors, the Akka library [5] supplies another implementation of non-strict actors for Scala. Besides offering better performance, it adds automatic load-balancing, improves the Erlang-style [3] resilience and fault-tolerance, and introduces opaque actor references for better encapsulation.

While making programming more convenient, breaking the actor abstraction has severe drawbacks. For example, most of the tools for testing [17,32], modeling [30], and verification [33] lose their efficiency when used on programs that mix the actor model with other concurrency models. Mixing concurrency models furthermore re-introduces the potential for fine-grained data races and reduces the readability and maintainability of programs.

3 Methodology

This section describes our methodology for compiling the corpus of Scala programs that use actors. It also explains how we gather statistics about these programs. We use these statistics to answer our research questions in Section 4 and complement them with discussions with the developers in Section 5.

3.1 The Corpus of Actor Programs

The foundation of our program corpus is the list of publicly available Scala programs on the github repository web site that import the Scala actor library or the Akka library. We ignore programs with less than 500 lines of code, which reduces the initial set of around 750 programs to a list of 270 programs[2]. Since the goal of the corpus is to characterize *real-world actor programs*, we further filter the list of 270 programs, reducing it to the 16 programs shown in Table 1. Our criteria for real-world actor programs are as follows:

(1) **Library Usage:** The program not only imports the Scala or Akka actor library, but also uses the library to implement a portion of its functionality. Note that this does not mean that the program uses the Actor class from the library. We merely require that it uses functionality provided by the library, for example futures or remote actors.
(2) **Size:** Scala makes it easy to import Java libraries and write programs containing a mixture of Java and Scala code. Since our analysis tool is agnostic to the difference, we require that the program consists of at least 3000 lines of code in total. Of these, at least 500 lines must be Scala code, which is more compact than Java code.
(3) **Eco-System:** At least two developers contribute to the project, and these provide a way to contact them. Intuitively, having two developers reduces the chance of completely random design mistakes; the agreement of two developers is more likely to represent a systematic mistake. Furthermore, the source code must compile, and the program must have documentation.

3.2 Data Collection

To collect the data underlying our statistics, we wrote a custom tool for analyzing the selected programs. The tool employs the WALA static analysis framework [10] and accepts the compiled bytecode of a program as input. It scans each class in the class hierarchy of the application, looking for indicators of relevant properties. For example, if the application contains a class that implements the actor base trait, it flags the application as making use of actors. Moreover, the tool detects the use of different communication mechanisms by scanning for the signatures of the relevant methods. For asynchronous communication in Scala actor programs, for example, it looks for the signature of `!` in `scala.actors.OutputChannel`.

[2] `http://actor-applications.cs.illinois.edu/index.html`

Table 1. *The corpus of actor applications studied in this paper.* The *Library* column denotes which actor library in which version the application uses. The *kLoC* columns list the number of thousand lines of source code in the application, without comments or empty lines, and the figures in the *Dev.* column shows the number of people contributing to the project. Detailed information about the selected programs, including their versions, is available at: `http://actor-applications.cs.illinois.edu/selected.html`.

			kLoC		
Library	*Program*	*Description*	*Scala*	*Java*	*Dev.*
Akka 2.0	BlueEyes	Web framework	28.6	–	3
Akka 2.0	Diffa	Real-time data differencing	29.6	5.2	8
Akka 2.1	Evactor	Event processing framework	4.6	–	2
Akka 2.0	Gatling	Stress test tool	8.2	0.6	19
Akka 2.0	GeoTrellis	Geographic data engine	10.1	–	2
Akka 2.0	Scalatron	Multi-player programming game	10.5	–	2
Akka 2.1	SignalCollect	Graph processing framework	4.6	–	4
Akka 2.0	Socko	Web server	5.7	1.6	5
Akka 2.0	Spray	RESTful web services library	15.8	–	8
Scala 2.9	BigBlueButton	Web conferencing system	0.8	52.5	30
Scala 2.7	CIMTool	Modeling tool	3.6	26.5	3
Scala 2.10	ENSIME	Scala interaction mode for emacs	8.0	–	19
Scala 2.9	Kevoree	Distributed systems platform	31.5	39.8	9
Scala 2.9	SCADS	Distributed storage system	26.3	1.0	15
Scala 2.9	Spark	Cluster computing system	12.2	–	17
Scala 2.9	ThingML	Modeling language for distributed systems	8.9	61.1	6

We chose this approach to overcome the lack of type and inheritance information in a purely string-based source code analysis. It allows us to gather data with higher precision in the following two situations: First, while programmers typically adhere to the convention of giving actor classes a name ending in `Actor`, this is not enforced by the compiler. Hence, a class `B extends A` that inherits from a class `A extends Actor` is an actor class that cannot be discovered through string matching. Second, because Scala employs type inference, programmers may—and often do—supply only a limited number of type annotations. This makes it hard to discover, for instance, whether an object or class uses a `Lock` for synchronization of concurrent operations. Aside from inheritance, other reasons for preferring Java bytecode over source code were the available tool set, and the ability to easily include mixed Scala–Java programs in our study.

The drawback of analyzing bytecode is the reduced precision of the results: the compiler discards (from its perspective) superfluous static information, which is thus no longer available to our tool. Consequently, some cases that could have been detected during compilation may now go unnoticed. However, the property detection methods we use are sound; our results are therefore lower bounds. In Section 4, we explain the detection mechanisms in detail.

4 Results

In this section, we answer our research questions with statistical data gathered from the programs in our corpus.

4.1 RQ1: How Often Do Scala Programmers Mix Actors with other Kinds of Concurrent Entities?

The Scala and Akka actor libraries provide two main constructs for implementing concurrent entities: Actor and Future. Futures are place-holders for asynchronously computed values; they block the current thread of execution when it tries to access a yet unresolved value. This way, futures provide a light form of synchronization between the producer and the consumer of the value. With their blocking result-resolution semantics, futures provide a natural way of adding partial synchrony to actor programs. Furthermore, Scala supports access to the Java standard library, which provides more conventional constructs for concurrent computation such as Runnable and Future (from java.util.concurrent). The programs in our corpus can therefore employ a mixture of actors, futures, and threads (via Runnable) to implement concurrent entities.

As an example, Listing 1 shows a code snippet from ENSIME, one of the programs in our corpus. In this code, DebugManager is an actor that has an inner class, MonitorOutput, which inherits from Runnable (Thread). When an instance of DebugManager is created, it initializes a list of MonitorOutputs, passing appropriate input streams to their constructors (Line 4). When DebugManager is started, it also starts the MonitorOutput threads in the list (Line 8), which leads to the execution of the run method in the MonitorOutputs (Line 24). Once a MonitorOutput is started, in a while loop, it reads data from the input stream and sends a message to project, which is itself an actor (Line 30). The DebugManager actor uses the finished variable (Line 23) to interrupt MonitorOutputs and break the while loop if desired (Line 29).

This example shows the mix of actors and threads in a single program. Our tool detects the elements of such a mixture of concurrent entities by checking for usage of Actor, Future, and Runnable. Recall that our tool does not count the usage of concurrent entities in the actor libraries, but only in the program itself. The use of the actor concurrency paradigm is detected via subclassing: if the application contains a class that implements the actor base trait, the tool marks the application as making use of actors[3]. Similarly, it detects thread-based concurrency through subclasses of java.lang.Runnable. Since application classes rarely inherit from futures, the tool detects the use of this concurrent entity by looking for fields, parameters, or local variables whose type is (a subtype of) Future.

[3] We furthermore include a detector for actors created by one of Scala's actor factory functions.

```
class DebugManager(project: Project, indexer: Actor,[...]) extends Actor {
  [...]

  private val monitor = List(new MonitorOutput([...]),new MonitorOutput([...]))

  def start() {
     [...]
     monitor.map { _.start() }
     [...]
  }
  def act() {
     loop {
     [...]
       receive {
         [...]
       }
     }
  }
  [...]

  private class MonitorOutput(val inStream: InputStream) extends Thread {
    val in = new InputStreamReader(inStream)
    @volatile var finished = false
    override def run() {
      try {
        var i = 0
        val buf = new Array[Char](512)
        i = in.read(buf, 0, buf.length)
        while (!finished && i >= 0) {
          project ! AsyncEvent(toWF(DebugOutputEvent(new String(buf, 0, i))))
          i = in.read(buf, 0, buf.length)
        }
      } catch {
        case t: Throwable => {
          t.printStackTrace()
        }
      }
    }
  }
}
```

Listing 1. A code snippet from ENSIME; see `https://github.com/aemoncannon/ensime/blob/d96f4e61ee85a07665348cb3933db7423082b428/src/main/scala/org/ensime/server/DebugManager.scala`

Table 2. *The usage of concurrency constructs.* A bullet (•) in the respective column means that the program contains a class that is derived from Actor; that is derived from Runnable; or that contains a field, parameter, or local variable whose type is (a subtype of) Future.

Program	*Actor*	*Runnable*	*Future*	*Program*	*Actor*	*Runnable*	*Future*
BlueEyes	•	•	•	BigBlueButton	•	•	–
Diffa	•	•	•	CIMTool	•	•	–
Evactor	•	–	–	ENSIME	•	•	–
Gatling	•	–	–	Kevoree	•	•	•
GeoTrellis	•	–	•	SCADS	–	•	•
Scalatron	•	•	•	Spark	•	•	•
SignalCollect	•	–	•	ThingML	•	–	–
Socko	•	•	–				
Spray	•	•	•				

Observations. The results in Table 2 show that 13 of the 16 programs (81%) mix concurrent entities and 12 of the 15 programs (80%) mix Actor with Runnable or Future. Specifically, the results indicate that futures alone seem to be insufficient to handle the concurrency related tasks of the programs: none of the programs relies solely on futures.

The use of futures together with actors has been long established and can be found in actor languages as early as ABCL [35]. Following this tradition, the Scala and Akka actor libraries support a special asynchronous messaging primitive for actors that returns a future. It is therefore not surprising to find that 8 out of 15 programs (53%) that use actors also use futures.

In Table 2, 10 out of 15 programs (66%) use both Actor and Runnable. The reasons for mixing Actor and Runnable are unclear. For example, while in Listing 1, the MonitorOutput could be an actor, which brings consistency to the concurrency model, it inherits from Runnable. One hypothesis would be that the program development started with thread-based concurrency, and later on shifted towards actors. However, by manually inspecting the programs and asking the developers for clarification, we discovered that this hypothesis is wrong. We discuss the details of our findings in Section 5.

4.2 RQ2: How Many of the Programs are Distributed over the Network, and Does Distribution Influence the Way Programmers Mix Concurrency Models?

The previous section shows that mixing the actor model with other concurrency models is common, and that the reason is not historical evolution. Another explanation could be that some models excel at a particular kind of programming. While the actor model allows both exploiting local processing resources, and distributing the program over the network, threads and shared-memory communication are limited to exploiting local processing resources.

As our second question, we ask whether this difference in support for distributed programming could be the reason for mixing concurrency models. Maybe programmers use actors for distributing the program over the network, but prefer threads for using the locally available processing resources on a multi-core machine.

Both actor libraries enforce the use of a special remote actor API in the case of network distribution. Our analysis tool can therefore distinguish between local and remote actor usage. Table 3 shows the results of searching the application code for invocations of the remote actor API.

Table 3. *The usage of actors for distributed programming.* Distributed programs spread their computation across a network; a bullet (•) in the *Distributed* column marks these. The *Remote* column shows which of the programs contain a class that uses the remote actor facilities provided by the actor library.

Program	*Remote*	*Distributed*	*Program*	*Remote*	*Distributed*
BlueEyes	–	–	BigBlueButton	–	•
Diffa	–	–	CIMTool	–	–
Evactor	–	–	ENSIME	–	–
Gatling	–	–	Kevoree	–	•
GeoTrellis	•	•	SCADS	–	•
Scalatron	–	–	Spark	•	•
SignalCollect	•	•	ThingML	–	–
Socko	–	•			
Spray	–	–			

Observations. Only 3 out of 16 programs use actors for remote deployment. This indicates that most developers use actors to address the local scalability problem, that is, they use actors as a solution for local concurrent programming.

However, we expected more of the applications to be distributed. To identify which of the applications are actually distributed (not necessarily using remote actors), we inspected the program code and contacted developers for confirmation. We found that 7 out of the 16 programs are distributed. This implies that *developers tend to use other ways than remote actors for implementing distributed computations.*

In Section 5 we discuss the reasons the developers gave for preferring other methods of distribution.

4.3 RQ3: How Often Do the Actors in the Programs Use Communication Mechanisms other Than Asynchronous Messaging?

The results of the previous section indicate that distributed computing is not the reason for mixing the actor model with other concurrency models. Consequently,

programmers seem to use actors to exploit local computing resources and achieve local scalability. If scalability is the goal, then, as motivated in Section 2, actors should communicate solely via asynchronous messages.

The limitation to asynchronous message-passing helps maintain scalability and avoid data races and deadlocks, but it adds complexity to coordination.

Instead of implementing asynchronous distributed protocols to achieve a coordination task, programmers can follow a different route to solve the coordination problems with Scala and Akka actors. In simple cases, programmers can use the provided synchronous messaging operations for blocking *remote procedure call* operations. Another option for synchronization are futures (see Section 4.1). Finally, Scala and Akka allow programmers to take a third route: programmers can choose to break the actor abstraction and rely on customary coordination methods from the shared-memory model, for example shared locks or latches.

An example of communication via a shared variable is shown in Listing 1. As mentioned in Section 4.1, the DebugManager actor uses a shared variable finished to communicate with each OutputMonitor thread. Listing 2 shows an example of a different way of communication: via latches. The code snippet is taken from Gatling. In the code, the Terminator actor coordinates several DataWriter actors—which are stored in the registeredDataWriters variable—to flush their data to the output stream when all of the users have finished their execution. The Terminator contains a latch which is initialized to the latch object sent in the Initialize message. The message's userCount argument determines the number of users in the program. The Terminator starts in the uninitialized state (Line 41), in which it can only accept Initialize messages (Line 13). After receiving an Initialize message, it changes to the initialized state, in which it can accept RegisterDataWriter and EndUser messages (Line 22). Upon receiving an EndUser message, the Terminator checks if the number of messages received from the users has reached the expected number. If that is the case, it creates futures that ask the DataWriter actors to flush their data to the output stream (Line 31). After all futures have completed their tasks, the Terminator counts down the latch (Line 35) to notify the entity that waits on the latch about the completion of its task. Therefore, the latch serves as communication channel between the Terminator actor and other entities in the program.

While using communication mechanisms other than asynchronous messaging can solve coordination and memory-limitation problems, breaking the actor abstraction re-introduces problems that actor semantics carefully avoid: shared state between actors allows fine-grained data races and breaks transparent distribution; blocking and synchronous operations can lead to deadlocks and, for older versions of the Java standard library, exhaust the available threads in the threadpool implementations of Scala and Akka.

Hence, using actors with communication mechanisms other than asynchronous messaging is a trade-off decision. Our third question asks how common these trade-offs are. We consider three main categories of communication:

```
class Terminator extends BaseActor {

  import context._

  /**
   * The countdown latch that will be decreased when all message are written and all
       scenarios ended
   */
  private var latch: CountDownLatch = _
  private var userCount: Int = _

  private var registeredDataWriters: List[ActorRef] = Nil

  def uninitialized: Receive = {

    case Initialize(latch, userCount) =>
      this.latch = latch
      this.userCount = userCount
      registeredDataWriters = Nil
      context.become(initialized)
  }

  def initialized: Receive = {

    case RegisterDataWriter(dataWriter: ActorRef) =>
      registeredDataWriters = dataWriter :: registeredDataWriters

    case EndUser =>
      userCount = userCount - 1
      if (userCount == 0) {
        implicit val timeout = Timeout(configuration.timeOut.actor seconds)
        Future.sequence(registeredDataWriters.map(_.ask(FlushDataWriter).mapTo[
            Boolean]))
          .onComplete {
            case Left(e) => error(e)
            case Right(_) =>
              latch.countDown
              context.unbecome
          }
      }
  }

  def receive = uninitialized
}
```

Listing 2. A code snippet from Gatling; see `https://github.com/excilys/gatling/blob/974299f78c433dcc7f4a1a46501127a41c37e11c/gatling-core/src/main/scala/com/excilys/ebi/gatling/core/result/terminator/Terminator.scala`

(1) *Non-blocking operations* like sending asynchronous messages (sm); resolving a future (rf); and signaling a synchronization construct (ss), for example counting down a latch or releasing a lock.
(2) *Blocking operations* like waiting to receive a message from a channel (wm); waiting for a future to be resolved (wf); and waiting for a synchronization construct to be signaled (ws), for example waiting on a latch.
(3) *Other operations* that do not fit in either of the above categories, for example communication via external resources like files or shared objects that are not synchronization constructs.

To answer RQ3, our tool searches through all Actor classes in each program, detecting if a class—or any of its super-classes—uses the non-blocking or blocking communication operations described in categories (1) or (2). The tool finds instances of non-blocking or blocking communications by looking for the signatures of the relevant methods. For example, for asynchronous communication in Scala actor programs it looks for invocations of library-defined methods like that of `!` in `scala.actors.OutputChannel`.

If the tool finds an actor class that does not use any of the blocking or non-blocking communication methods, it puts it in the *other* category. However, because of the static nature of our analysis, our tool may not be able to detect indirect use of blocking or non-blocking operations. For example, suppose class ServiceActor is an actor class that has a field, printer, instantiated from class Printer. The Printer class is not an actor class but has a method print that performs asynchronous messaging (non-blocking operation). If ServiceActor *only* communicates by invoking printer.print, then our tool does not diagnose ServiceActor as using non-blocking operations and hence marks it as belonging to the *other* category. Therefore, actor classes in the *other* category may indirectly use blocking or non-blocking operations. To address this problem, we manually inspected the classes reported as using *other* category to confirm that they do not use any of the communication mechanisms from the blocking or non-blocking categories.

By using the above method, as mentioned in Section 3, we obtain a lower bound on the usage of each kind of communication operation. Note that a more precise detection of each kind of communication operation requires a more complex analysis [22] and is beyond the capability of our tool. We consequently leave this task for future work.

The results are shown in Table 4. We removed SCADS because it does not use Actors from the libraries. For every program in Table 4, we mark a communication mechanism with a bullet (•) if we find at least one Actor in the program that uses that mechanism. Otherwise, we mark it with an en-dash (–). Consequently, we mark a program with a bullet for *other* if we find at least one Actor that does not use any of the six blocking or non-blocking communication operations in category (1) or (2).

Observations. As the results show, 2 out of 15 programs (ENSIME and Kevoree) use blocking operations to receive a message. Moreover, two programs (Gatling and BlueEyes) contain an Actor that communicates through non-blocking operations on futures or synchronization constructs. However, 6 of the 15 programs

Table 4. *Communication of actors with other entities.* A bullet (•) denotes that the program contains an actor that uses a communication mechanism in the respective category. The operations in the *Non-blocking* column are sending asynchronous messages (sm); resolving a future (rf); and signaling a synchronization construct (ss) like a latch. *Blocking* operations are waiting on a message from a channel (wm); waiting for a future to be resolved (wf); and waiting for a synchronization construct (ws). *Other* operations do not fit either of these categories.

Program	*Non-block.*			*Blocking*			*Other*	*Program*	*Non-block.*			*Blocking*			*Other*
	sm	rf	ss	wm	wf	ws			sm	rf	ss	wm	wf	ws	
BlueEyes	•	•	–	–	–	–	–	BBButton	•	–	–	–	–	–	–
Diffa	•	–	–	–	–	–	•	CIMTool	•	–	–	–	–	–	–
Evactor	•	–	–	–	–	–	•	ENSIME	•	–	–	•	–	–	–
Gatling	•	–	•	–	–	–	–	Kevoree	•	–	–	•	–	–	•
GeoTrellis	•	–	–	–	–	–	–	Spark	•	–	–	–	–	–	–
Scalatron	•	–	–	–	–	–	–	ThingML	–	–	–	–	–	–	•
SignalCollect	•	–	–	–	–	–	•								
Socko	–	–	–	–	–	–	•								
Spray	•	–	–	–	–	–	–								

(40%) contain an Actor that *only* communicates via an operation of the *other* category. Manual inspection of the actors using *other* communication reveals that in two programs, the actors perform I/O, and in four programs the actors operate on a shared object. In these cases, developers were willing to accept the potential drawbacks of data races and deadlocks to solve the problem at hand.

Recall that these numbers might be lower than the actual values. For example, an actor like DebugManager in Listing 1 uses asynchronous messaging as well as operations of the *other* category (shared object). However, since the tool finds that the actor uses asynchronous messaging, it does not report the actor as using communication from *other* category.

To summarize, *in at least 9 out of 15 programs (60%), actors use communication mechanisms other than asynchronous messaging.*

5 The Reasons for Mixing Concurrency Models

The results presented in Section 4 show that around 68% (11 out of 16) of the real-world Scala actor programs in our corpus mix Runnable with actor library constructs like Actor and Future. To investigate the reasons, we manually inspected these programs and contacted the developers, asking them about the details of their design decisions. In order to avoid a bias toward a specific reason, we omitted potential answers. Instead, we posed open-ended questions of the form: *Why did you implement module X with Runnable and not with Actor?*

We received answers from the developers of 10 programs (all 11 programs except CIMTool). After receiving the answers, we dismissed the initial hypothesis that the programs started with thread-based concurrency that was later (partially) replaced with actors or futures: only for 3 of the 11 programs did the

answers indicate such a motivation. In the cases of the other eight programs, the developers desired to have pure actor programs. However, they faced problems and as a consequence decided to replace Actor with Runnable. We categorize the reasons into three groups:

- *Actor library inadequacies*: The reasons in this category are lack of library support for efficient I/O, as well as problems implementing low-end systems, managing many blocking operations, and customizing the default features of the actor implementation.
- *Actor model inadequacies*: Certain protocols are easier to implement using shared-memory than using asynchronous communication mechanisms without shared state.
- *Inadequate developer experience*: Developers lack enough knowledge about the library, or reuse legacy code.

5.1 Actor Library Inadequacies

Efficient I/O. The developers of four programs mention efficient input and output (I/O) management as a reason for using Runnable. They either decided to perform I/O in dedicated threads (and not to use threads from the actor library thread pool) to avoid deadlock cases, or they claimed efficiency and performance benefits through dedicated I/O threads.

BigBlueButton: The developers of BigBlueButton use Runnables for reading and writing I/O streams to avoid blocking actors which are executed on the library thread pool. However, they agree that it is possible to refactor the Runnables into actors running on a specific thread[4]. While the Scala actor documentation describes a pattern for this use case [11], it seems that the pattern is either too obscure or inconvenient to implement.

Spark: Spark is a distributed computation framework that needs to exchange large blocks of data over the network. Because the developers are unsure about the actor library's performance regarding large data transfer, they spawn dedicated threads for handling this task.

> "[...] in ParallelShuffleFetcher, we are receiving large blocks of data from multiple machines. Most actor libraries don't deal well with that – they are optimized for transferring small messages (up to a few hundred bytes) [...], and they might have a small number of IO threads that block when you're sending something bigger. In this case, instead of worrying about whether the actor library will handle the transfer well [...] and whether it will affect other messages being sent by other actors, we chose to explicitly spawn threads. I'd love an actor library that also handles large IOs, or exposes asynchronous IO primitives, but I haven't found one."

[4] `https://groups.google.com/forum/?fromgroups=#!topic/bigbluebutton-dev/2ad-HBeNQeY`

The above message by one of the developers shows that there is demand for an API that gives programmers control over the I/O operations. Moreover, it shows the lack of documentation of the Scala actor library's capabilities: because the capabilities of the library are *unknown*, the developers chose a *known* solution using Runnables, accepting the design drawbacks.

Spray: The Spray framework builds upon the Akka library, which, unlike the Scala actor library, provides an API for managing I/O. Despite this, the Spray developers implement a custom module for asynchronous network I/O using Runnable. As motivation, the Spray developers explain[5] that tailoring the implementation to the specific use case yields performance benefits over using the (more abstract) API of Akka. This is confirmed by one of the Akka developers.

ENSIME: In ENSIME, multiple Runnables are created and executed to read and write from I/O streams. One of the developers expressed that, since there is no need for the actor mail box, using Runnable has less overhead.

Low-End Systems. Mobile phones and low-end systems are among the target platforms of the Kevoree framework for dynamically reconfigurable distributed systems. While Kevoree uses Scala actors, it uses threads to implement the core components that are shared among all platforms. The developers state performance considerations as their motivation:

> "[...] JVM ForkAndJoin implementation and other implementation of Thread are very slow and switching context cost a lot of computational power. Again part of Core section of Kevoree are now write with thread to avoid such limitations. [...] More globally this is true for the whole Scala library which is now growing more and more.[...] Porting such a library of more than 10 mb is now challenging for limited environment (RaspberryPI) and especially for Android, it was a nightmare [...]"

Although there are some actor libraries specialized for writing embedded systems [8], Scala actors are not a proper solution for embedded systems.

Managing and Debugging Many Blocking Operations. The Kevoree middleware also contains parts with many blocking operations. According to explanations by the developers, some operations define atomic actions which should block the calling thread and wait for the completion of the operation. Initially, the developers started with a pure actor-based solution in which actors used blocking operations for receiving messages. However, they faced deadlock problems and decided to replace the blocking actors with threads:

> "In earlier versions of Kevoree we used Actor everywhere [...]. That was a mistake because we faced a lot of deadlock use cases. Some deadlock was issues from bug in the ForkAndJoin implementation in JVM

[5] `https://groups.google.com/d/msg/spray-user/b4YwS5XUsB8/8q_88qs2Gu0J`

> and some others went from OS limitation (for example using VPS hosting which limited the number of process). For those reasons critical section of Kevoree now start with some dedicated threads, which costs a little more but is far easier to manage in case of blocking actors."

The developers also mention that Runnable helped them to manage blocking operations:

> "[...] we use plain old thread when dealing with third party library which do some waiting operation internally. Using thread let us to control such blocking operation and allow use to start a sibling watchdog thread when something goes wrong. We could also use ThreadedActor but in this case the benefit is not so important."

As noted by the developers, it is possible to execute an actor in a dedicated thread (ThreadedActor) and manage such blocking cases. However, the developers decided to follow the old way of programming. In fact, when facing problems with actors, the developers replaced some of them with Runnables to debug the program. After finding the root cause of the problems, they decided to stay with Runnables so that they can handle similar problems more easily in the future.

The explanations given by the developers indicates that the abstraction that actor libraries provide over threads complicates conventional debugging approaches.

Customized Actors. The developers of SCADS and BlueEyes implemented their own actor-like entities using Runnable and Future.

SCADS: The SCADS distributed database system aims at improving performance with their customized actors. The problem faced by the developers was that the Scala actor library uses a hard-coded serializaion mechanism (the default Java serializer) when sending messages over the network. To make use of a more efficient serialization mechanism, the developers implemented custom actor-like concurrent objects. These objects are furthermore optimized towards processing the key–value messages customary to the program. While Akka provides an API to customize the serialization of messages, this library was not tried by the developers of SCADS.

BlueEyes: The developers of BlueEyes are interested in having *typed actors*. Neither Scala, nor default Akka actors use type information to characterize the messages that an actor accepts or sends. Hence, the compiler cannot discover whether an actor sends the wrong type of message to another actor. To have support for this kind of static composition checks, the BlueEyes developers implemented their own actor-like class hierarchy. The classes incorporate in their signature the types of messages that are acceptable for the actor, and the types of messages sent by the actor.

5.2 Actor Model Inadequacies

The developers of BlueEyes found using actors to implement the coordination protocol in their HTTP server harder than implementing the protocol with threads.

> "Now let's look at Actors. They address concurrency and mutual exclusion, but they conflate the two (you either get both or none). They don't address coordination at all – you have to build your own protocols for coordination. This code [...] is all about coordination, so using a lock is much simpler way to implement it than using an Actor."

The problem pointed out by the developers concerns purely asynchronous systems in general and is not restricted to the Akka or Scala actor libraries. To give an intuition of this problem, consider the example shown in Listing 3. The DataProcessor processes an array of data supplied to its processData method and returns an array of results. The results should be ordered such that the value in results(i) corresponds to dataArray(i).

For the sake of performance, data processing is implemented in parallel using the fork–join pattern [20]: for each element in dataArray, a future is created that processes the element and puts the result in the results array. Since the results data structure is shared between the futures, it is protected by a synchronized block. The algorithm waits for the results to become ready by calling awaitAll on the futures and returns the results to the caller.

An alternative implementation of the same algorithm using Scala actors and purely asynchronous communication is shown in Listing 4. In this code, the DataProcessor is an actor that, upon receiving ProcessData message (Line 10), stores administrative information in its local variables and then delegates the processing to worker actors. The administrative information consists of the results array, the customer reference to the sender of the message, the totalCount of data elements in dataArray, and curCount, which is the number of results received so far. For each data element in the dataArray, the DataProcessor creates a Worker actor and sends it a Process message.

Unlike in the previous implementation, to preserve encapsulation, the results array is not shared between the worker actors. Instead, workers send the results to the DataProcessor via ProcessResult messages (Line 36) and let the DataProcessor put them in the results array (Line 22).

There are two issues that need to be resolved with this solution. The first issue is that, because of asynchrony, the DataProcessor may receive the results in an arbitrary order from the worker actors. To address this issue, each Process message not only contains the data element to process, but also its index. The worker actors send the results using the same index in ProcessResult messages. This allows the DataProcessor actor to order the results.

The second issue is that the DataProcessor must know when the results are ready to be sent to the customer. This is addressed through the totalCount and curCount variables. The variable curCount is incremented after each ProcessResult message (Line 23). When it reaches totalCount, the DataProcessor knows that

```
1  class DataProcessor {
2
3    def processData(dataArray: Array[Data]): Array[Result] = {
4
5      var results = new Array[Result](data.length)
6
7      val workers = for (i <− 0 to data.length − 1) yield
8        future {
9          var r = process(data(i)) //process data
10         synchronized {
11           results(i) = r
12         }
13       }
14
15     awaitAll(20000L, workers: _*)
16     return results
17   }
18
19 }
```

Listing 3. Implementation of a parallel data processor in the shared-memory model

processing is complete and sends the results to the customer (Line 24). Note that the customer variable, which records the sender of the ProcessData message, is needed to return the results to the right client. Unlike in synchronous method invocation, there is no implicit return address.

The example shows that implementing some coordination protocols in the actor model can be more complex than using a shared-memory model. The developers may need to add extra variables and implement more complex logic to handle the asynchrony in the actor model that is not present in the shared-memory model. Specifically, for developers who are new to the actor model, understanding and managing coordination in an asynchronous and no-shared state model might be harder than in the shared-memory model.

To address this problem, prior work has extended the Scala actor library with coordination patterns used in parallel programming, for example joins [12] and divide-and-conquer tasks [15]. More advanced coordination mechanisms for actor systems have also been proposed [4,31,9,27]. However,to the best of our knowledge, none has been integrated with a widely used actor library.

5.3 Inadequate Developer Experience

In three programs, Socko, Scalatron, and Diffa, the developers did not have any special objection to the actor library. They used Runnable because (1) they used to their traditional style of programming; (2) they had some legacy code and wanted to reuse it; or (3) they did not want to trust a new technology when

```
1  class DataProcessor extends Actor {
2
3    var curCount = 0
4    var totalCount = 0
5    var results: Array[Result] = _
6    var customer: OutputChannel[Any] = _
7
8    def act() = loop {
9      react {
10       case ProcessData(dataArray: Array[Data]) => {
11         results = new Array[Result](dataArray.length)
12         customer = sender
13         totalCount = dataArray.length
14         curCount = 0
15
16         for (i <- 0 to totalCount - 1) {
17           var worker = new Worker().start()
18           worker ! Process(dataArray(i), i)
19         }
20       }
21       case ProcessResult(result: Result, index: Int) => {
22         results(index) = result
23         curCount += 1
24         if (curCount == totalCount) customer ! results
25       }
26     }
27   }
28 }
29
30 class Worker extends Actor {
31
32   def act() = loop {
33     react {
34       case Process(data: Data, index: Int) => {
35         var r = process(data(i)) //process data
36         sender ! ProcessResult(r, index)
37       }
38     }
39   }
40
41 }
```

Listing 4. Implementation of a parallel data processor in the actor model

using Runnable would be enough for implementing the required functionality. They use actors when it is necessary to handle concurrent accesses to an object. In these cases, having automated tools that can detect such inconsistencies and help developers to transform the Runnable to Actor would be helpful.

6 Implications and Discussion

In this section, we combine our analysis results with feedback from the developers to give recommendations to researchers and library designers.

Implications for Researchers. The analysis results show that mixing concurrency models is common in real-world Scala programs that use actor libraries. Each model has its strengths, and developers tend to use the model that best fits the problem. However, the current implementations of actors in the Scala standard library and Akka force developers to use models other than actors to meet the application requirements.

On the one hand, research on modeling, testing, and analysis tools for actor programs should take this into account. Specifically, mixtures of Actor and Future are common, as they help implementing coordination between the purely asynchronous actors. Therefore, unless the proposed tools and approaches for actor programs can handle a mixture of actors with other concurrent entities, only few real-world programs can benefit from them.

On the other hand, the results show that in three cases, mixing actors with threads is unnecessary. Automated tools that can detect such cases and help developers refactor threads to actors in their programs would alleviate the problem of mixing concurrency models.

The actor model itself also puts a burden on developers. The property of no shared state and asynchronous communication can make implementing coordination protocols harder than using established constructs like locks. However, providing a language for coordination protocols would alleviate this problem.

Implications for Library Designers. The library APIs can help developers comply with the best practices of a concurrency model in two ways:

- First, the API can provide commonly required features like modules for efficiently handling or customizing I/O. This would address one of the main problems that Scala developers currently have in pure actor programs.
- Second, it can prevent developers from misusing the library constructs and violating best practices. For example, if messages were restricted to immutable types, actors could not easily share objects by exchanging references through messages. While libraries cannot completely prevent shared state in actors, such a limitation would push developers towards using a proper design.

Apart from the API, library-specific tools for debugging and testing would be beneficial for developers. In particular, the high-level abstraction of actors makes

it hard for developers to trust, test, and debug low-level execution. A way to get insight into the execution mechanism would reduce these worries.

Finally, clarifying the limitations and capabilities of the libraries also helps developers make the right decisions during the design and development of their programs.

7 Threats to Validity

Internal threats to the validity of this study concern the accuracy of our data collection tool. An inherent limitation is its use of static analysis: the detected method invocations and usage of concurrency constructs may not represent the usage at run time. Moreover, since our tool cannot detect indirect method invocations, the reported statistics about the communication operations may be lower than the actual values. To alleviate this problem, we supplemented the analysis with manual inspection when the tool could not detect any kind of our targeted communication operations. For the cases that the tool reported the usage of concurrency constructs or communication operations, we randomly selected some reported instances and confirmed the results by manual inspection.

To ensure that the concluded reasons for mixing the actor model with other concurrency models are aligned with the real reasons, we eliminated any bias towards specific answers in our questions to the developers. Moreover, we validated the reasons supplied by the developers against the library documentation and other related resources. We discuss our findings after each developer answer in Section 5.

The external threats are related to how much our results are generalizable. To ensure external validity regarding other Scala actor programs, we (1) obtained our programs from github, which we found to be the most common repository site for Scala programs by surveying the Akka and Scala mailing lists; and (2) target the two most popular actor libraries for Scala.

Our selection criteria (Section 3.1) exclude the majority of programs from the initial list, which greatly shrinks the sample size. However, the criteria ensure high-quality specimens by preventing the inclusion of programs with overly idiosyncratic styles of single programmers and test projects. The criteria also exclude large enough programs that we could not compile; however, only four programs were excluded on this ground. Finally, we compiled our initial list of programs one year ago. Consequently, it will exclude programs hosted only recently on github. Since we demand a certain maturity of projects, we do not expect this to be problematic.

The actor libraries we target have features similar to many other actor libraries for imperative languages [26,7,16], which also allow mixing threads and actors. We therefore believe that our results hold for actor programs written with these libraries. However, the results may not hold for actor languages like Erlang [3] that put more restrictions on the language constructs to force programmers to comply with the foundations of the actor model.

8 Related Work

To the best of our knowledge, this is the first systematic study of the phenomenon of mixing concurrency models in a single program.

The work most closely related to our study are comparisons between different libraries and paradigms for multi-core programming. They are controlled user studies that aim to determine the productivity of programmers. Nanz et al. [21] compare two object-oriented languages, multi-threaded Java and SCOOP, for concurrent programming. Besides productivity, the comparison also focuses on the correctness of the programs written by the participants. Luff [19] compares three concurrent programming paradigms: the actor model, transactional memory, and standard shared-memory threading with locks, in Java. Pankratius et al. [25] compare Scala as an imperative and functional language with Java as an imperative language for concurrent programming. None of these studies considers the mixing of concurrency models.

Several empirical studies investigate the usage of the concurrency constructs from a single library. Naturally, these studies are confined to the concurrency model of the library and do not discuss mixed models. Weslley et al. [34] study 2000 Java projects to determine the most commonly used Java concurrent library constructs. They also analyze usage trends over time. Similarly, Okur and Dig [24] study programs using the Microsoft parallel libraries. By analyzing programs semantically, they achieve higher precision than the syntactic analysis of Weslley et al. The study of Hochstein et al. [14] concerns the productivity of developers using MPI in a large-scale project.

Other studies [18,6] collect and document common mistakes in the usage of concurrent constructs in a single library that lead to concurrency bugs. These collections help developers and researchers to prevent them in the future. However, they are also confined to the concurrency model of the library.

Another line of work integrates the actor model with task parallelism. Haller et al. [12] augment the Scala actor library with join patterns. PAM [29] adds parallel execution of messages inside of actors to achieve better performance. JCoBox [28] combines actors and futures to implement parallel execution of tasks and synchronous messaging. Immam et al. [15] propose a unified parallel programming model for Scala and Java that integrates the actor model with the divide-and-conquer task parallel model. These works use small benchmarks to show that implementing certain protocols with their proposed model is easier and can provide better performance than the pure actor model. However, none of these works conducts any study on real-world programs to show the weaknesses of the actor libraries or the actor model. Our study complements these works by supplying the empirical evidence for these weaknesses.

9 Conclusion and Future Work

This study is the first to investigate how often and why developers mix the actor model with other concurrency models, which has severe drawbacks (Section 2).

The study uses a corpus of real-world Scala programs collected from public github repositories (Section 3). Statically analyzing the programs reveals (Section 4) that most of the programs (80%) mix actors with other concurrent entities. 66% of the programs combine actors and threads, and 53% combine actors and futures. Moreover, at least 60% of all programs contain an actor that does not communicate via asynchronous messaging. Thus, in some situations, other factors than the advantages of asynchronous message-passing dominate the decisions of developers. Through discussion with the developers (Section 5), we find that the reasons for mixing concurrency models and avoiding asynchronous communication lie in inadequacies of the actor libraries and the actor model itself. In Section 6, we discuss the implications of our findings for researchers and library designers.

A direction for future work is to correlate the phenomenon of mixing concurrency models with bug rates and types. This would reveal whether the phenomenon we observed is actually a problem, and if so, which concurrency constructs may help to remedy it. A related question is whether mixing occurs across different layers of abstraction. For example, mixing may occur only on the lower, more concrete layers of the program while actors prevail on the higher, more abstract layers. Finally, it would be interesting to see how different actor libraries for the same language, for example Scala, affect the design decisions of programmers. Results from such investigation would provide guidance for the library developers.

Acknowledgments. The authors would like to thank Nicholas Chen, Stas Negara, Marjan Sirjani, Yun Young Lee, Minas Charalambides, Philipp Haller, Viktor Klang, and Madan Musuvathi for their comments on the earlier versions of this paper. This paper is based upon work partially supported by the U.S. Department of Energy under Grant No. DOE DE-FG02-06ER25752, and the Army Research Office under award W911NF-09-1-0273. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
2. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997)
3. Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD thesis, Kungl Tekniska Högskolan (2003), `http://www.erlang.org`
4. Atkinson, R., Hewitt, C.: Synchronization in actor systems. In: Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1977, pp. 267–280 (1977)
5. Bonér, J., Klang, V., Kuhn, R., et al.: Akka library, `http://akka.io`
6. Bradbury, J.S., Jalbert, K.: Defining a catalog of programming anti-patterns for concurrent Java. In: Proc. of the 3rd International Workshop on Software Patterns and Quality, SPAQu 2009, pp. 6–11 (October 2009)

7. Bykov, S., Geller, A., Kliot, G., Larus, J.R., Pandya, R., Thelin, J.: Orleans: cloud computing for everyone. In: Proc. of the 2nd ACM Symposium on Cloud Computing, SOCC 2011, pp. 16:1–16:14 (2011)
8. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 31–40 (2005)
9. Dinges, P., Agha, G.: Scoped synchronization constraints for large scale actor systems. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 89–103. Springer, Heidelberg (2012)
10. Dolby, J., Fink, S.J., Sridharan, M.: T. J. Watson libraries for analysis (WALA), `http://wala.sf.net`
11. Haller, P., Sommers, F.: Actors in Scala. Artima Series (2012)
12. Haller, P., Van Cutsem, T.: Implementing joins using extensible pattern matching. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 135–152. Springer, Heidelberg (2008)
13. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proc. of the 3rd International Joint Conference on Artificial Intelligence, IJCAI 1973, pp. 235–245 (1973)
14. Hochstein, L., Shull, F., Reid, L.B.: The role of MPI in development time: a case study. In: Proc. of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 34:1–34:10 (2008)
15. Imam, S.M., Sarkar, V.: Integrating task parallelism with actors. In: Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 753–772 (2012)
16. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: Proc. of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, pp. 11–20 (2009)
17. Lauterburg, S., Dotta, M., Marinov, D., Agha, G.: A framework for state-space exploration of Java-based actor programs. In: Proc. of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, pp. 468–479 (2009)
18. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGPLAN Not. 43(3), 329–339 (2008)
19. Luff, M.: Empirically investigating parallel programming paradigms: A null result. In: PLATEAU at the ACM Onward! Conference (2009)
20. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming, 1st edn. Addison-Wesley Professional (2004)
21. Nanz, S., Torshizi, F., Pedroni, M., Meyer, B.: Design of an empirical study for comparing the usability of concurrent programming languages. In: Proc. of the 2011 International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, pp. 325–334 (2011)
22. Negara, S., Karmani, R.K., Agha, G.: Inferring ownership transfer for efficient message passing. In: Proc. of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 81–90 (2011)
23. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, 2/e. Artima Series. Artima Press (2010)
24. Okur, S., Dig, D.: How do developers use parallel libraries? In: Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012, pp. 54:1–54:11 (2012)

25. Pankratius, V., Schmidt, F., Garretón, G.: Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. In: Proc. of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 123–133 (2012)
26. Pech, V., König, D., Winder, R., et al.: GPars, `http://gpars.codehaus.org/`
27. Proença, J., Clarke, D., de Vink, E., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: Proc. of the 27th Annual ACM Symposium on Applied Computing, SAC 2012, pp. 1510–1515 (2012)
28. 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)
29. Scholliers, C., Tanter, E., Meuter, W.D.: Parallel actor monitors. In: 14th Brazilian Symposium on Programming Languages (2010)
30. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 20–56. Springer, Heidelberg (2011)
31. Song, M., Ren, S.: Coordination operators and their composition under the actor-role-coordinator (ARC) model. SIGBED Rev. 8(1), 14–21 (2011)
32. Tasharofi, S., Gligoric, M., Marinov, D., Johnson, R.: Setac: A framework for phased deterministic testing of Scala actor programs. In: The Second Scala Workshop (2011)
33. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE 2012. LNCS, vol. 7273, pp. 219–234. Springer, Heidelberg (2012)
34. Torres, W., Pinto, G., Fernandes, B., Oliveira, J.A.P., Ximenes, F.A., Castor, F.: Are Java programmers transitioning to multicore?: a large scale study of Java FLOSS. In: SPLASH Workshops, SPLASH 2011 Workshops, pp. 123–128 (2011)
35. Yonezawa, A. (ed.): ABCL: an object-oriented concurrent system. MIT Press, Cambridge (1990)

Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-Order Library

Kasper Svendsen[1], Lars Birkedal[1], and Matthew Parkinson[2]

[1] IT University of Copenhagen
{kasv,birkedal}@itu.dk
[2] Microsoft Research Cambridge
mattpark@microsoft.com

Abstract. We present a case study of formal specification for the C♯ joins library, an advanced concurrent library implemented using both shared mutable state and higher-order methods. The library is specified and verified in HOCAP, a higher-order separation logic extended with a higher-order variant of concurrent abstract predicates.

1 Introduction

It is well-known that modular specification and verification of concurrent higher-order imperative programs is very challenging. In the last decade good progress has been made on reasoning about subsets of these programming language features. For example, higher-order separation logic with nested triples has proved useful for modular specification and verification of higher-order imperative programs that use state with little sharing, e.g., [22,16,15]. Nested triples support specification of higher-order methods and higher-order quantification allows library specifications to abstract over the internal state maintained by the library and the state effects of function arguments.

Likewise, concurrent abstract predicates [7] has proved useful for reasoning about shared mutable data structures in a concurrent setting. Concurrent abstract predicates (CAP) extends separation logic with protocols governing access to shared mutable state. Thus CAP supports modular specification of shared mutable data structures that abstract over the *internal sharing*, e.g., [5]. However, CAP does not support modular reasoning about *external sharing* – the sharing of *other* mutable data structures through a shared mutable data structure. For instance, CAP does not support modular reasoning about locks[1] – the canonical example of a shared mutable data structure used to facilitate external sharing.

We have recently proposed HOCAP [26], a new program logic which combines higher-order separation logic with concurrent abstract predicates and extends concurrent abstract predicates with *state-independent higher-order protocols*. To reason about external sharing through a data structure, we parameterise the specification of the data structure with assertions that clients can instantiate to

[1] See Section 6 for a discussion of this issue.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 327–351, 2013.

describe the resources they wish to share through the data structure. Higher-order protocols allow us to impose protocols on these external resources when reasoning about the implementation of the data structure. State-independent higher-order protocols allow us to reason about non-circular external sharing patterns.

HOCAP is thus intended as a general purpose program logic for modular specification and verification of concurrent higher-order imperative programs with support for modular reasoning about *both* internal and external sharing. We have previously verified simple examples in HOCAP. In this paper we report on an extensive case study of a sophisticated and realistic library that combines all these challenges in one, to test whether HOCAP can in fact be used to give an abstract formal specification.

In particular, we explore how to give a modular specification to a concurrent library that features internal sharing and is used to facilitate external sharing. Clients interact with the library using reentrant callbacks. The specification should thus abstract over the internal state while allowing abstract reasoning about external sharing through the library and reentrant calls back into the library. Furthermore, the specification should of course be strong enough to reason about clients, and weak enough to allow the implementation of the library to be verified against the specification.

Our case study of choice is the C♯ joins library [20]. The joins library, which is based on the join calculus [8,9], provides a declarative concurrency model based on message passing. Declarative message patterns are used to specify synchronisation conditions and function arguments are used to specify synchronisation actions. Synchronisation actions might themselves cause new messages to be sent, leading to reentrant callbacks. The joins concurrency model is useful for defining new synchronisation primitives – i.e., to facilitate external sharing. Finally, the library itself is implemented using internal state.

In this paper we present a formal specification of a subset of the C♯ joins library in HOCAP. The specification is expressed in terms of the high-level join primitives exposed by the library and hides all internal state from clients. Moreover, we test the specification of the joins library by verifying a number of synchronisation primitives for which there are already accepted specifications in the literature. For example, we verify that a reader-writer lock implemented using joins can be proved to satisfy the standard separation logic specification for a reader-write lock. We have chosen to focus on synchronisation primitives because synchronisation primitives are specifically designed to facilitate external sharing.

In addition to its role as a case study of a higher-order reentrant concurrent library, the specification of the joins library is itself interesting. The main idea behind the specification is to allow clients of the joins library to impose ownership transfer protocols at the level of the join primitives exposed by the library. As illustrated with several examples, this leads to natural and short proofs of synchronisation primitives implemented using the joins library.

We have also verified a simple lock-based implementation of the joins library. However, in this paper we focus on the joins specification and the use thereof,

since the main point is to investigate how HOCAP can be used to give abstract specifications for concurrent higher-order imperative libraries. We refer the interested reader to the accompanying technical report for details about the verification of the joins implementation [24]. This paper does not require the reader to understand all the details of HOCAP.

Outline. The remainder of the paper is organised as follows. In Section 2 we give an extensive introduction to the joins library using a series of examples to explain each feature of the library. Along the way, we sketch how one can reason informally, in separation-logic style, about the correctness of the applications. In Section 3 we summarise the necessary bits of HOCAP. This leads us to Section 4, where we introduce the formal specification of the joins library. In Section 5 we revisit a couple of the example applications and show how the informal proof sketches from Section 2 can be turned into formal proofs using the formal specification from Section 4. Finally, we evaluate and discuss the case study in Section 6.

2 Introducing Joins

The joins concurrency model is based on the concept of messages, which are used both for synchronisation and communication between threads. Conceptually, a join instance consists of a single message pool and a number of *channels* for adding messages to this pool. Channels come in two varieties, *synchronous* and *asynchronous*. Sending a message via a synchronous channel adds the message to the message pool and blocks the sender until the message has been received. Asynchronous channels simply add messages to the message pool, without blocking the sender.

The power of the joins calculus stems from how messages are received. One declares a set of *chords*, each consisting of a *pattern* (a condition on the message pool) and a continuation. When a pattern matches a set of messages in the message pool, the chord may *fire*, causing the continuation to execute. Crucially, once a chord fires, the messages that matched the pattern are removed from the message pool *atomically*, making them unavailable for future matches. Upon termination of the continuation, the clients that added the removed messages via synchronous channels are woken up and allowed to continue. We say that a message has been *received* when it has been matched by a chord and the chord continuation has terminated.

In the rest of this section we introduce the C# joins library, one feature at a time. Each new feature is introduced with a joins example of a synchronisation primitive implemented using this feature. For each example, we sketch an informal proof of the synchronisation primitive in separation logic. The examples thus serve both to introduce the joins library and motivate the main ideas behind our formal specification of the joins library.

We take as a starting point Russo's joins library for C$^\sharp$ [20], with a slightly simplified API. In particular, we have omitted value-carrying channels, as value-carrying channels do not add any conceptual difficulties.

2.1 Synchronous Channels

Sending a message via a synchronous channel causes the sender to block until the message has been received. To illustrate, we consider the example of a 2-barrier – an asymmetric barrier restricted to two clients.

Implementation. One can implement a 2-barrier as a joins instance with two synchronous channels – one for each client to signal its arrival. Clients should block at the barrier until both clients have signalled their arrival. This can be achieved with a single chord with a pattern that allows it to fire when there is a pending message on both channels (i.e., when both clients have arrived). The C♯ code for a 2-barrier is given in Figure 1.

The TwoBarrier constructor creates a join instance, j, and two synchronous channels, ch1 and ch2, attached to the underlying message pool of this join instance. Next, the constructor creates a pattern p that matches any pair of messages in the message pool consisting of a ch1 message (i.e., a message added via the ch1 channel) and a ch2 message. Lastly, it registers this pattern as a chord without a continuation. Hence, this chord may fire when there is a pending message on both channels and when it fires, it atomically removes and receives these two messages from the message pool. Each Arrive method signals the client's arrival by sending a message on the corresponding channel using the Call method.

```
class TwoBarrier {
  private SyncChannel ch1;
  private SyncChannel ch2;

  public TwoBarrier() {
    Join j = new Join();
    ch1 = new SyncChannel(j);
    ch2 = new SyncChannel(j);
    Pattern p = j.When(ch1).And(ch2);
    p.Do();
  }

  public void Arrive1() { ch1.Call(); }
  public void Arrive2() { ch2.Call(); }
}
```

Fig. 1. Joins 2-barrier implementation

All the examples we consider in this article follow the same structure as the above example: the constructor creates a join instance with accompanying channels and registers a number of chords. After this initialisation phase, the chords and channels stay fixed and interaction with the joins instance is limited to the sending of messages.

We now sketch a proof of this 2-barrier implementation using separation logic. Recall that separation logic assertions, say P and Q, describe and assert ownership of resources and that $P * Q$ holds if P and Q describe (conceptually) disjoint resources. The logic will be introduced in greater detail in Section 4 when we get to the formal specification and formal reasoning.

Desired Specification. From the point of view of resources, a barrier allows clients to exchange resources. We call these resources external as they are typically external to the barrier data structure itself. On arrival at the barrier each client may transfer ownership of some resource to the barrier, which is then redistributed atomically once both clients have arrived. For the purpose of this introduction we will make the simplifying assumption that each client transfers

the same resource to the barrier on each arrival and that these resources are redistributed in the same way at each round of synchronisation. In Section 5.2 we consider a general specification without these simplifying assumptions.

Under these assumptions we can specify the barrier in terms of two predicates, B_i^{in} and B_i^{out}, where B_i^{in} describes the resources client i transfers to the barrier upon arrival, and B_i^{out} describes the resources client i expects to receive back from the barrier upon leaving. These predicates thus describe the external resources clients intend to share through the barrier. Since a barrier can only redistribute resources (i.e., it cannot create resources out of thin air), the combined resources transferred to the barrier must imply the combined resources transferred back from the barrier: $B_1^{\mathsf{in}} * B_2^{\mathsf{in}} \Rightarrow B_1^{\mathsf{out}} * B_2^{\mathsf{out}}$.

The *client* of the barrier is thus free to pick any B_i^{in} and B_i^{out} predicates satisfying the above redistribution property. We can now express the expected specification of a 2-barrier b in terms of these abstract predicates:

$$\{B_1^{\mathsf{in}}\}\ \mathsf{b.Arrive1}()\ \{B_1^{\mathsf{out}}\} \qquad\qquad \{B_2^{\mathsf{in}}\}\ \mathsf{b.Arrive2}()\ \{B_2^{\mathsf{out}}\}$$

That is, for client 1 to arrive at the barrier (i.e., to call $\mathsf{Arrive1}$), it has to provide the resource described by B_1^{in}, and if the call to $\mathsf{Arrive1}$ terminates (i.e., client 1 has left the barrier), it will have received the resource described by B_1^{out}.

Proof Sketch. The main idea behind our specification of the joins library is to allow clients to impose an ownership transfer protocol on messages. An ownership transfer protocol consists of a *channel precondition* and a *channel postcondition* for each channel. The channel precondition describes the resources the sender is required to transfer to the recipient when sending a message on the channel. The channel postcondition describes the resources the recipient is required to transfer to the sender upon receiving the message.

In the 2-barrier example, sending a message on a channel corresponds to signalling one's arrival at the barrier. The channel preconditions of the barrier thus describe the resources clients are required to transfer to the barrier upon their arrival. Hence, we take each channel precondition to be the corresponding B^{in} predicate: $P_{\mathsf{ch1}} = B_1^{\mathsf{in}}$ and $P_{\mathsf{ch2}} = B_2^{\mathsf{in}}$. Throughout this section we use the notation P_{ch} to refer to the channel precondition of channel ch and Q_{ch} to refer to the channel postcondition.

The barrier implementation features a single chord that matches and receives both arrival messages, once both clients have arrived. The channel postconditions of the barrier thus describes the resources the barrier is required to transfer back to the clients, once both clients have arrived. We thus take each channel postcondition to be the corresponding B^{out} predicate: $Q_{\mathsf{ch1}} = B_1^{\mathsf{out}}$ and $Q_{\mathsf{ch2}} = B_2^{\mathsf{out}}$.

One can thus think of the channel pre- and postconditions as specifications for channels. Since the channel postcondition describes the resources transferred back to the sender *once* its message has been received, one should think of it as a partial correctness specification. In particular, without any chords to receive messages on a given channel we can pick any channel postcondition, as no message sent on that channel will ever be received. Conversely, whenever we add a new chord we have to prove that it satisfies the chosen ownership

transfer protocol. For chords without continuations, this reduces to proving that the preconditions of the channels that match the chord's pattern imply the postconditions of these channels.

The 2-barrier consists of a single chord that matches any pair of messages consisting of a ch1 message and a ch2 message. Correctness thus reduces to proving $P_{\mathsf{ch1}} * P_{\mathsf{ch2}} \Rightarrow Q_{\mathsf{ch1}} * Q_{\mathsf{ch2}}$, which follows from the assumed redistribution property.

2.2 Asynchronous Channels

The previous example illustrated the use of synchronous channels that block the sender until its message has been received. The joins library also supports *asynchronous* channels, allowing messages to be sent without blocking the sender. A lock is a simple example that illustrates the use of both asynchronous and synchronous channels. Acquiring a lock must wait for the previous thread using the lock to finish: it is synchronous. However, releasing a lock need not wait for the next thread to attempt to acquire it: it is asynchronous.

Implementation. We can implement a lock using the joins library as follows:

We use two channels acq and rel to represent the two actions one can perform on a lock. The join instance has a single chord with a pattern that matches any pair of messages consisting of an acq message and a rel message. Thus, to acquire the lock, a thread sends a message on the acq channel; the call will block until the chord fires, which can only happen if there is a rel message in the message pool. The lock is thus unlocked if and only if there is a pending rel message in the message pool. The release can happen asynchronously; it does not have to wait for the next thread to attempt to acquire the lock.

The lock is initially unlocked by calling rel.

```
class Lock {
  private SyncChannel acq;
  private AsyncChannel rel;

  public Lock() {
    Join j = new Join();
    acq = new SyncChannel(j);
    rel = new AsyncChannel(j);
    j.When(acq).And(rel).Do();
    rel.Call();
  }

  public void Acquire() { acq.Call(); }
  public void Release() { rel.Call(); }
}
```

Fig. 2. A Joins implementation of a lock

Desired Specification. Locks are used to ensure exclusive access to some shared resource. We can specify a lock in separation logic in terms of an abstract resource predicate R (picked by the client of the lock) as follows:

$$\{R\}\ \mathsf{new\ Lock()}\ \{\mathrm{emp}\} \qquad \{\mathrm{emp}\}\ \mathsf{l.Acquire()}\ \{R\} \qquad \{R\}\ \mathsf{l.Release()}\ \{\mathrm{emp}\}$$

When the lock is unlocked the resource described by R is owned by the lock. Upon acquiring the lock, the client takes ownership of R, until it releases the lock again. Since the lock is initially unlocked, creating a new lock requires ownership of R to be transferred to the lock. This is the standard separation logic

specification for a lock [17,10,11]. Here R thus describes the external resources shared through the lock.

Proof Sketch. Informally, we can understand the rel message as moving the resource protected by the lock from the thread to the join instance, and the acq message as doing the converse. This can be stated more formally using channel pre- and postconditions as follows: $P_{\mathsf{acq}} = \text{emp}$, $Q_{\mathsf{acq}} = R$, $P_{\mathsf{rel}} = R$, and $Q_{\mathsf{rel}} = \text{emp}$.

Recall that channel postconditions describe the resources the recipient is required to transfer to the sender upon receiving the message. Since the sender of a message on an asynchronous channel has no way of knowing if its message has been received, channel postconditions do not make sense for asynchronous channels. We thus require channel postconditions for asynchronous channels to be empty, emp.

As before, to prove that the acq and rel chord satisfies the channel postconditions, we have to show that the combined channel preconditions imply the combined channel postconditions: $P_{\mathsf{acq}} * P_{\mathsf{rel}} \Rightarrow Q_{\mathsf{acq}} * Q_{\mathsf{rel}}$. This follows directly from the fact that $*$ is commutative.

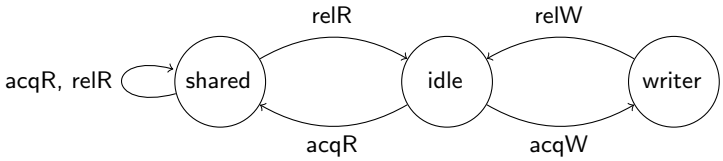
2.3 Continuations

So far, every chord we have considered has simply matched and removed messages from the message pool. In general, a chord can have a continuation that is executed when the chord fires, before any blocked synchronous senders are allowed to continue.

Continuations can, for instance, be used to automatically send a message on a certain channel when a chord fires. Thus they can be used to encode a state machine. Moreover, one can also ensure that a state of the state machine is correlated with the history or state of the synchronisation primitive that one is implementing. To illustrate, we extend the lock from the previous example into a biased reader/writer lock.[2]

A reader/writer lock [4] generalises a lock by introducing read-only permissions. This allows multiple readers to access a shared resource concurrently. To determine whether a read or write access request should be granted, three states suffice: (idle) no readers or writers, (writer) exactly one writer, or (shared) one or more readers. In the idle state there are no readers or writers, so it is safe to grant both read and write access. In the shared state, as one client has already been granted read access, it is only safe to grant read access. We can express this as a state machine as follows:

[2] Biased here means that this reader/writer implementation may starve the writer thread. It is possible to extend this implementation into an unbiased reader/writer lock by introducing an additional asynchronous channel to distinguish between whether or not there are any pending writers when a reader request has been granted.



Here acqR and acqW refers to the acquire read and acquire write operation, and relR and relW refers to the release read and release write operation.

Implementation. The idea is to encode this state machine using three asynchronous channels, idle, shared, and writer, with the invariant that there is at most one pending asynchronous message in the message pool at any given time. This gives a direct encoding of the three states in the above state machine, and adds a fourth intermediate state (when there is no pending message on any of the three asynchronous channels). The intermediate state is necessary for the implementation, as it does not transition atomically between the states of the above state machine. The joins implementation is given below.

```
class RWLock {
  private SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel idle, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join j = new Join();
    // ... initialise channels ...

    j.When(acqR).And(idle).Do(AcqR);
    j.When(acqR).And(shared).Do(AcqR);
    j.When(relR).And(shared).Do(RelR);
    j.When(acqW).And(idle).Do(writer.Call);
    j.When(relW).And(writer).Do(idle.Call);

    idle.Call();
  }

  private void AcqR() {
    readers++;
    shared.Call();
  }

  private void RelR() {
    if (--readers == 0)
      idle.Call();
    else
      shared.Call();
  }

  public void AcquireR() { acqR.Call(); }
  public void AcquireW() { acqW.Call(); }
  public void ReleaseR() { relR.Call(); }
  public void ReleaseW() { relW.Call(); }
}
```

We use three asynchronous channels to encode the current state in the above state machine and thus to determine whether a read or write access can be granted. In addition, we use the readers field to count the actual number of readers, to determine which state to transition to when releasing a reader. Note that the continuation given to Do is a named C♯ delegate, and that in all five cases, the given continuation sends a message on an asynchronous channel. These calls are *reentrant* calls back into the joins library, making these continuations *reentrant callbacks*.

Note further that all five chords consume exactly one asynchronous message and sends exactly one asynchronous message. Between consuming and sending the asynchronous message, there are no pending asynchronous messages and the reader/writer is in the previously mentioned fourth state. Hence, between

consuming and sending an asynchronous message, no other chord can fire and the currently executing continuation has exclusive access to the internal state of the reader/writer lock (i.e., the readers field).

Desired Specification. The standard separation logic specification for a reader/writer lock is expressed using counting permissions [2]. Counting permissions allow a full write permission to be split into any number of read permissions, counting the total number of read permissions, to allow them to be joined up to a full write permission later. The standard specification is given below in terms of an abstract resource predicate for writing to the resource R_{write} and an abstract resource predicate for reading the resource R_{read}.

$$\begin{array}{rll} \{R_{write}\} & \textbf{new RWLock()} & \{\text{emp}\} \\ \{\text{emp}\} & \text{l.AcquireR()} & \{R_{read}\} \\ \{\text{emp}\} & \text{l.AcquireW()} & \{R_{write}\} \\ \{R_{read}\} & \text{l.ReleaseR()} & \{\text{emp}\} \\ \{R_{write}\} & \text{l.ReleaseW()} & \{\text{emp}\} \end{array} \tag{1}$$

To avoid introducing counting permissions directly, we specify the reader/write lock in terms of an additional family of abstract resource predicates $R(n)$, indexed by $n \in \mathbb{N}$, satisfying that $R(0)$ is the full write permission R_{write}, and $R(n)$ is the permission left after splitting off n read permissions. Thus R should satisfy, $\forall n \in \mathbb{N}.\ R(n) \Leftrightarrow R_{read} * R(n+1)$ and $R(0) \Leftrightarrow R_{write}$. Note that a client of the reader/writer lock is free to pick any R_{write}, R_{read} and R that satisfies these two properties.

Proof Sketch. The three asynchronous channels encode the current state of the reader/writer lock. The channel preconditions of the three asynchronous channels thus describe the resources owned by the reader/writer lock in the idle, shared and writer state, respectively. In the idle state (no readers or writers), the reader/writer lock owns the readers field and the full write permission, and the readers field contains 0. In the shared state (one or more readers), the reader/writer lock owns the readers field and the remaining permission after splitting off n read permissions and the readers field contains n. Lastly, in the writer state (exactly one writer), the writer owns the full resource and the reader/writer lock only owns the readers field.

$$P_{\mathsf{idle}} = \mathsf{readers} \mapsto 0 * R(0) \qquad P_{\mathsf{writer}} = \mathsf{readers} \mapsto 0$$
$$P_{\mathsf{shared}} = \exists n \in \mathbb{N}.\ n > 0 * \mathsf{readers} \mapsto n * R(n)$$

Since idle, shared, and writer are asynchronous, their channel postconditions must be empty (as explained earlier).

For the synchronous channels we can read off their channel pre- and postconditions directly from the desired specification (1):

$$P_{\mathsf{acqR}} = \text{emp} \qquad Q_{\mathsf{acqR}} = R_{read} \qquad P_{\mathsf{acqW}} = \text{emp} \qquad Q_{\mathsf{acqW}} = R(0)$$
$$P_{\mathsf{relR}} = R_{read} \qquad Q_{\mathsf{relR}} = \text{emp} \qquad P_{\mathsf{relW}} = R(0) \qquad Q_{\mathsf{relW}} = \text{emp}$$

To register a chord without a continuation we had to show that the combined channel preconditions *implied* the combined channel postconditions. What about

the present case with a proper continuation? Since the continuation runs before the release of any blocked synchronous callers, we have to show that the continuation *transforms* the combined channel preconditions to the combined channel postconditions. For the reader/writer lock we thus have to show the proof obligations on the left in Figure 3. These proof obligations are all completely standard and mostly trivial separation logic proofs. For instance, the proof of the first obligation is given on the right in Figure 3. Note that in this proof we use the

$$
\begin{array}{cccc}
\{P_{\mathsf{acqR}} * P_{\mathsf{idle}}\} & \mathsf{AcqR}() & \{Q_{\mathsf{acqR}} * Q_{\mathsf{idle}}\} \\
\{P_{\mathsf{acqR}} * P_{\mathsf{shared}}\} & \mathsf{AcqR}() & \{Q_{\mathsf{acqR}} * Q_{\mathsf{shared}}\} \\
\{P_{\mathsf{relR}} * P_{\mathsf{shared}}\} & \mathsf{RelR}() & \{Q_{\mathsf{relR}} * Q_{\mathsf{shared}}\} \\
\{P_{\mathsf{acqW}} * P_{\mathsf{idle}}\} & \mathsf{writer.Call}() & \{Q_{\mathsf{acqW}} * Q_{\mathsf{idle}}\} \\
\{P_{\mathsf{relW}} * P_{\mathsf{writer}}\} & \mathsf{idle.Call}() & \{Q_{\mathsf{relW}} * Q_{\mathsf{writer}}\}
\end{array}
$$

$$
\begin{array}{l}
\{P_{acqR} * P_{idle}\} \\
\{readers \mapsto 0 * R(0)\} \\
\quad \mathsf{readers{+}{+};} \\
\{readers \mapsto 1 * R(0)\} \\
\{readers \mapsto 1 * R_{\mathsf{read}} * R(1)\} \\
\quad \mathsf{shared.Call();} \\
\{R_{\mathsf{read}}\} \\
\{Q_{acqR} * Q_{idle}\}
\end{array}
$$

Fig. 3. Left: Proof obligations for the reader/writer lock chords. Right: A proof sketch for the first proof obligation of the reader/writer lock.

channel pre- and postcondition of the shared channel. These proofs thus have a similar character to partial correctness proofs of a recursive method, where one is allowed to assume the specification of a method while proving that its body satisfies the assumed specification. Here we assume the shared channel obeys the chosen ownership transfer protocol while proving that the first chord obeys the chosen protocol.

2.4 Nonlinear Patterns

The public interface of the 2-barrier in Section 2.1 is slightly non-standard, as it has two distinct arrival methods. A more standard barrier interface would provide a common Arrive method, for both clients. The joins library also supports an implementation of a barrier with such an interface, through the use of *nonlinear patterns*. Nonlinear patterns match multiple messages from the same channel.

Implementation. We can thus implement a more standard 2-barrier as a joins instance with a single synchronous arrival channel and a single chord with a nonlinear pattern that matches two messages on the arrival channel. Clearly this generalises to an n-barrier, which can be implemented as follows.

```
class Barrier {
  private SyncChannel arrive;

  public Barrier(int n) {
    Join j = new Join(); arrive = new SyncChannel(j); Pattern p = j.When(arrive);
    for(int i = 1; i < n; i++) { p = p.And(arrive); };
    p.Do();
  }
```

```
  public void Arrive() { arrive.Call(); }
}
```

This code registers a single chord with a pattern that matches n messages on the synchronous arrive channel.

Desired Specification. As before, assume predicates B_i^{in} and B_i^{out} (picked by the client), where B_i^{in} describes the resources client i transfers to barrier upon arrival and B_i^{out} describes the resources client i expects to receive back from the barrier upon leaving. These predicates should satisfy the following redistribution property, $\circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{in}} \Rightarrow \circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{out}}$, to allow the barrier to redistribute the combined resources, once every client has arrived.

From the informal description of B_i^{in} and B_i^{out} one might thus expect an n-barrier b to satisfy the following specification:

$$\{B_i^{\mathsf{in}}\}\ \mathsf{b.Arrive()}\ \{B_i^{\mathsf{out}}\}$$

That is, if a client transfers B_i^{in} to the barrier upon arrival, it should receive back B_i^{out} from the barrier upon leaving. However, this specification is not quite right. In particular, what prevents client i from impersonating client j when it arrives at the barrier? To apply the redistribution property to the combined resources transferred to the barrier we need to ensure that when client i arrives at the barrier, it actually transfers B_i^{in} to the barrier, even if it also happens to own B_j^{in}. Hence, while the barrier *implementation* no longer distinguishes between clients, we still need a way to distinguish clients *logically*. We can achieve this by introducing a client identity predicate, $\mathsf{id}(i)$ to assert that the owner is client i. By making this predicate non-duplicable, we can enforce that clients cannot impersonate each other.

We can now express a correct barrier specification in terms of this id predicate as follows:

$$\{\mathrm{emp}\}\ \mathbf{new}\ \mathsf{Barrier(n)}\ \{\circledast_{i\in\{1,\ldots,\mathsf{n}\}}\mathsf{id}(i)\} \quad \{B_i^{\mathsf{in}} * \mathsf{id}(i)\}\ \mathsf{b.Arrive()}\ \{B_i^{\mathsf{out}} * \mathsf{id}(i)\}$$

Upon creation of a new n-barrier we get back n id assertions. These are then distributed to each client to witness their identity when they arrive at the barrier.

Proof Sketch. Our proof sketch of the 2-barrier in Section 2.1 exploited that the implementation used distinct channels to signal the arrival of each client, which allowed us to pick different channel pre- and postconditions for each client. Since the above implementation uses a single arrival channel we have to pick a common channel pre- and postcondition that works for every client. We can achieve this using a *logical argument* to relate the channel precondition and the channel postcondition. In this case we index the channel pre- and postcondition with the client identifier i: $P_{\mathsf{arrive}}(i) = B_i^{\mathsf{in}} * \mathsf{id}(i)$ and $Q_{\mathsf{arrive}}(i) = B_i^{\mathsf{out}} * \mathsf{id}(i)$.

For the id predicate to witness the identity of clients, it must be non-duplicable. That is, it must satisfy, $\mathsf{id}(i) * \mathsf{id}(j) \Rightarrow i \neq j$. To define the id predicate such that it satisfies the above property, we need to introduce a bit more of our logic. We return to this example in Section 5.2.

3 Logic

The program logic is a higher-order separation logic [1] with support for reasoning about concurrency, shared mutable data structures [7,6], and recursive delegates [22]. We use this one program logic to reason about both *clients* of the joins library, and an *implementation* of the joins library.

Our program logic is a general purpose logic for reasoning about higher-order concurrent C♯ programs. We have presented the logic in a separate paper [26]. The full logic and its soundness proof is included in the accompanying technical report [23] of that paper. For the present paper we limit our attention to those features necessary to verify our client examples. To this end, it suffices to consider a minor extension of higher-order separation logic with fractional permissions, phantom/auxiliary state and nested triples [21].

Higher-Order Separation Logic. Every specification in Section 2 was expressed in terms of abstract resource predicates, such as the lock invariant R. This is easily and directly expressible in a higher-order logic, by quantification over predicates [1,19].

Our assertion logic is an intuitionistic higher-order separation logic over a simply typed term language. The set of types is closed under function space and products, and includes the type of propositions, Prop, the type of specifications, Spec, and the type of mathematical values, Val. The Val type includes all C♯ values and strings, and is closed under formation of pairs, such that mathematical sequences and other mathematical objects can be conveniently represented.[3]

Fractional Permissions. The notion of ownership in standard separation logic is very limited, supporting only two extremes: exclusive ownership and no ownership. To formalise the examples from the previous section we need a middle ground of read-only permissions, which can be freely duplicated. Fractional permissions [3] provide a popular solution to this problem, by annotating the points-to predicate with a fraction $p \in (0,1]$, written $x.f \overset{p}{\mapsto} v$. Full permission corresponds to $p = 1$ and grants exclusive access to the field f. Permissions can be split and combined arbitrarily ($x.f \overset{p}{\mapsto} v * x.f \overset{q}{\mapsto} v \Leftrightarrow x.f \overset{p+q}{\mapsto} v$). Any fraction less than 1 grants partial read-only access to the field f. We write $x.f \mapsto v$ as shorthand for $x.f \overset{1}{\mapsto} v$ and $x.f \overset{-}{\mapsto} v$ as shorthand for $\exists p \in (0,1].\ x.f \overset{p}{\mapsto} v$.

Phantom State. Auxiliary variables [18] are commonly used as an abstraction of the history of execution and state in Hoare logics. Normally, one declares a subset of program variables as auxiliary variables that can be updated using standard variable assignments, but are not allowed to affect the flow of execution. To support this style of reasoning, we extend separation logic with phantom state. Like standard auxiliary variables, phantom state allows us to record an abstraction of the history of execution, but unlike standard auxiliary variables, phantom state is purely a logical construct (i.e., the operational semantics of the programming language is not altered to accommodate phantom state and phantom state is not

[3] We use a single universe Val for the universe of mathematical values to avoid also having to quantify over types in the logic. We omit explicit encodings of pairs and write $(v_1, ..., v_n)$ for tuples coded as elements of Val.

updated through programming level assignments). When combined with logical arguments, phantom state allows us to logically distinguish and relate multiple messages on the same channel, as needed for the n-barrier example.

Phantom state extends objects with a logical notion of phantom fields and an accompanying phantom points-to predicate, written $x_f \overset{p}{\mapsto} v$, to make assertions about the value and ownership of these phantom fields. To support read-only phantom fields, we further enrich the notion of ownership with fractional permissions. Thus $x_f \overset{p}{\mapsto} v$ asserts the ownership of phantom field f of object x, with fractional permission p, and that this phantom field currently contains the value v. Like the normal points-to predicate, phantom points-to satisfies $x_f \overset{p_1}{\mapsto} v_1 * x_f \overset{p_2}{\mapsto} v_2 \Rightarrow v_1 = v_2$ as phantom fields contain a single fixed value at any given point in time.

Phantom fields are updated using a *view shift*. The notion of a view shift comes from the Views framework for compositional reasoning about concurrency [6], and generalises assertion implication. A view shift from assertion p to assertion q is written $p \sqsubseteq q$. Views shifts can be applied to pre- and postconditions using the following generalised rule of consequence:

$$\frac{p \sqsubseteq p' \qquad \{p'\}c\{q'\} \qquad q' \sqsubseteq q}{\{p\}c\{q\}}$$

Given full ownership (fractional permission 1) of a phantom field f, one can perform a logical update of the field ($x_f \overset{1}{\mapsto} v_1 \sqsubseteq x_f \overset{1}{\mapsto} v_2$). To create a phantom field f we require that the field does not already exist, so that we can take full ownership of the field. We thus require all phantom fields of an object o to be created simultaneously when o is first constructed.

Figure 4 contains a selection of inference rules from our program logic, related to view shifts and phantom state.

Nested Triples. To reason about delegates we use nested triples [21]. We write $x \mapsto \{P\}\{Q\}$ to assert that x refers to a delegate satisfying the given specification.

$$\frac{p \Rightarrow q}{p \sqsubseteq q} \qquad \frac{p \sqsubseteq q}{p * r \sqsubseteq q * r} \qquad \frac{p \sqsubseteq p' \qquad \{p'\}c\{q'\} \qquad q' \sqsubseteq q}{\{p\}c\{q\}}$$

$$\frac{}{x_f \overset{1}{\mapsto} v_1 \sqsubseteq x_f \overset{1}{\mapsto} v_2} \qquad \frac{}{x_f \overset{p_1}{\mapsto} v_1 * x_f \overset{p_2}{\mapsto} v_2 \Rightarrow v_1 = v_2} \qquad \frac{}{x_f \overset{p}{\mapsto} v * x_f \overset{q}{\mapsto} v \Leftrightarrow x_f \overset{p+q}{\mapsto} v}$$

Fig. 4. Selected program logic inference rules

Reasoning about the Implementation. Fractional permissions introduce a more lenient ownership discipline that allows for read-only sharing. To verify the *implementation* of the joins library, we need even more general forms of sharing. To reason about general sharing patterns we base our logic on concurrent abstract predicates [7].

Conceptually, concurrent abstract predicates (CAP) partitions the heap into a set of regions that each come with a protocol governing how the state in that region may evolve. This allows *stable* assertions – assertions that are closed under changes permitted by the protocol – to be freely duplicated and shared. To ensure soundness, the logic requires that all pre- and postconditions in the specification logic are stable. We thus introduce a new type, SProp, of stable assertions.

Concurrent abstract predicates with first-order protocols (i.e., protocols that only refer to the state of their own region) suffice for reasoning about sharing of *primitive resources* such as individual heap cells. To reason about sharing of *shared resources* requires *higher-order protocols* that can relate the state of multiple regions. In general, to reason about sharing of shared resources requires reasoning about circular sharing patterns. HOCAP extends concurrent abstract predicates with a limited form of higher-order protocols – called state-independent higher-order protocols – and introduce partial orders to explicitly rule out these circular sharing patterns.

Since we are using the same program logic to reason about join clients and the underlying join implementation, join clients could themselves use CAP to describe shared resources when picking the channel pre- and postconditions. This could potentially introduce circular sharing patterns. To simplify the presentation and focus on the main ideas behind our specification of the joins library we have chosen to present a specification that *does not* allow clients to use CAP in their channel pre- and postconditions. This allows us to give a simple specification without any proof obligations about the absence of circular sharing patterns. In the accompanying technical report, we define a stronger joins specification that *does* allow clients to use CAP, but requires clients to prove the absence of circular sharing patterns. In the technical report we verify the joins implementation against this stronger specification. See Section 6 for further discussion.

To prevent joins clients from using CAP, we introduce a new type, LProp, of local propositions. Every predicate expressible in the language of higher-order separation logic extended with phantom state and nested triples is of type LProp, provided all higher-order quantifications quantify over LProp rather than Prop. However, LProp is not closed under region and action assertions for reasoning about shared mutable data structures using CAP. All assertions of type LProp are trivially stable and LProp is thus a subtype of SProp. We thus require all channel pre- and postconditions to be of type LProp. This ensures that clients do not introduce circular sharing patterns.[4]

For details about the logic see our HOCAP paper and accompanying technical report [26,23].

[4] This circular sharing pattern has been allowed by the first two authors recent work [25].

4 Joins Specification

In this section we present our formal specification for the joins library.

The full specification of the joins library is presented in Figure 5. To simplify the specification and exposition of the joins library, we require all channels and chords be registered before clients start sending messages.[5] Formally, we introduce three phases:

ch: This phase allows new channels to be registered.
pat: This phase allows new chords to be registered.
call: This phase allows messages to be sent.

A newly created join instance starts in the ch phase. Once all channels have been registered, it transitions to the pat phase. Once all chords have been registered, it transitions to the call phase. In the call phase, the only way to interact with the join instance is by sending messages on its channels.

The specification is expressed in terms of a number of abstract representation predicates. We use three join representation predicates, $\mathsf{join}_{\mathsf{ch}}$, $\mathsf{join}_{\mathsf{pat}}$ and $\mathsf{join}_{\mathsf{call}}$ – one for each phase – which will be explained below. In addition, we use two representation predicates for channels and patterns:

$\mathsf{ch}(c, j)$: This predicate asserts that c refers to a channel registered with join instance j.
$\mathsf{pat}(p, j, X)$: This predicate asserts that p refers to a pattern on join instance j that matches the multi-set of channels X.

These representation predicates are all existentially quantified in the specification; clients thus reasons abstractly in terms of these predicates.

Channel Initialisation Phase. In the first phase we use the join representation predicate: $\mathsf{join}_{\mathsf{ch}}(A, S, j)$. This predicate asserts that j refers to a join instance with asynchronous channels A and synchronous channels S.

The join constructor (Join) returns a new join instance in the ch phase with no registered channels.

The two rules for creating and registering new channels (Sync and Async) take as argument a join instance j in the ch phase and return a reference to a new channel. In both cases, we get back a ch assertion, ch(r,j), that asserts that this newly created channel is registered with join instance j. In addition, both postconditions explicitly assert that this newly created channel is distinct from all previously registered channels, $\mathsf{r} \notin A \cup S$. As the channel predicate is duplicable ($\mathsf{ch}(c, j) \Leftrightarrow \mathsf{ch}(c, j) * \mathsf{ch}(c, j)$), to allow multiple clients to use the same channel, we have to state this explicitly.

Chord Initialisation Phase. In the second phase we use the join representation predicate: $\mathsf{join}_{\mathsf{pat}}(P, Q, j)$. This predicate asserts that j refers to a join instance with channel preconditions P and channel postconditions Q. Here P and Q are functions that assign channel pre- and postconditions to each channel.

[5] This restriction rules out reasoning about self-modifying synchronisation primitives. We are not aware of any examples of self-modifying join clients.

To relate the pre- and postcondition of a channel (as needed, e.g., in the n-barrier example to distinguish clients), we index each channel pre- and postcondition with a logical argument of type Val.[6] Formally P and Q are thus functions of type $P, Q : \text{Val} \times \tau_{\text{chan}} \to \text{LProp}$ where τ_{chan} is the type of channel references.[7]

Once sufficient channels have been registered, the join instance can transition into the chord initialisation phase using a view shift:

$$\frac{\forall c, a.\ c \in A \Rightarrow Q(a, c) = \text{emp}}{\mathsf{join}_{\mathsf{ch}}(A, S, j) \sqsubseteq \mathsf{join}_{\mathsf{pat}}(P, Q, j)}$$

This forces all channel pre- and postconditions to be fixed before any chords can be registered. This rule explicitly requires that the channel postconditions of asynchronous channel are empty, emp, as explained in Section 2.2.

Rules When and And create a new singleton pattern, and add new channels to an existing pattern, respectively. Note that a pattern matches a multi-set of channels and the set-union in And is thus multi-set union.

The rules for Do are more interesting. Rule Do1 deals with patterns without a continuation. Recall from our informal proof sketches that to add a new chord without a continuation we showed that the combined channel preconditions of the chord pattern *implied* the combined channel postconditions. Our specification generalises this to require that the combined channel preconditions can be *view shifted* to the combined channel postconditions. This generalisation allows us to perform logical updates of phantom state when the chord fires. We will see why this is useful in Section 5.2.

Furthermore, since our channel pre- and postconditions are now indexed by a logical argument, we have to prove that we can perform this view shift for any logical arguments (we have a logical argument for each channel). Formally,

$$\forall Y \in \mathcal{P}_m(\text{Val} \times \tau_{\text{chan}}).\ \pi_{\mathsf{ch}}(Y) = X \Rightarrow \circledast_{y \in Y} P(y) \sqsubseteq \circledast_{y \in Y} Q(y)$$

where $\mathcal{P}_m(-)$ denotes the finite power multi-set operator and π_{ch} is the power set lifting of π_2. Y thus associates a logical argument with each channel. To register a chord that matches channels x and y, this thus reduces to two universally quantified logical arguments, say a and b:

$$\forall a, b \in \text{Val}.\ P(a, x) * P(b, y) \sqsubseteq Q(a, x) * Q(b, y)$$

The rule for Do with a continuation (Do2) is very similar, but instead of requiring a view-shift, it takes a delegate b that transforms the combined preconditions into the combined postconditions. Crucially, the delegate is given access to the join instance in the message phase. This enables it to send messages, as used in the reader/writer lock example (Section 2.3).

Message Phase. The final phase allows messages to be sent. We use a third abstract predicate, $\mathsf{join}_{\mathsf{call}}(P, Q, j)$, with the same parameters as the previous abstract predicate $\mathsf{join}_{\mathsf{pat}}(P, Q, j)$. Once all chords have been registered, the join

[6] As Val is closed under pairs this allows us to encode an arbitrary number of logical arguments of type Val.

[7] Formally, τ_{chan} is simply a synonym for Val, introduced to improve the exposition.

instance can transition into the third phase using a view shift: $\mathsf{join}_{\mathsf{pat}}(P, Q, j) \sqsubseteq \mathsf{join}_{\mathsf{call}}(P, Q, j)$.

The only operation in the third phase is to send messages using Call. The rule for sending a message is very similar to the standard method call rule: we provide the precondition $P(a, c)$ and get back the postcondition $Q(a, c)$. Here a is the logical argument, which the client is free to pick.

Both the $\mathsf{join}_{\mathsf{call}}$ and $\mathsf{ch}(-)$ predicate is freely duplicable, to allow multiple clients to send messages on the same channel:

$$\mathsf{ch}(c, j) \Leftrightarrow \mathsf{ch}(c, j) * \mathsf{ch}(c, j)$$
$$\mathsf{join}_{\mathsf{call}}(P, Q, j) \Leftrightarrow \mathsf{join}_{\mathsf{call}}(P, Q, j) * \mathsf{join}_{\mathsf{call}}(P, Q, j)$$

Reasoning about Joins. We have verified a simple lock-based implementation of the joins library (see the accompanying technical report for details). We have thus given concrete definitions of the abstract predicates pat, ch, $\mathsf{join}_{\mathsf{ch}}$, $\mathsf{join}_{\mathsf{pat}}$, $\mathsf{join}_{\mathsf{call}}$ and proved that the implementation satisfies a generalisation of the joins specification in Figure 5.

5 Reasoning with Joins

In this section we revisit the lock and the n-barrier example, and sketch their formal correctness proofs in terms of our formal specification of the joins library. The lock example is intended to illustrate the joins specification in general, and has thus been written out in full. The n-barrier example is intended to illustrate the use of logical arguments and phantom state.

5.1 Lock

We begin by formalising the previous informal lock specification. As mentioned in Section 3, to avoid reasoning about sharing of shared mutable data structures through themselves, we require all channel pre- and postconditions to be local assertions – i.e., assertions of type LProp. Since the channel pre- and postconditions are defined in terms of the lock resource invariant, the lock resource invariant must be a local assertion. The formal specification of the lock is thus:

$$\forall R : \mathrm{LProp}.\ \exists\, \mathsf{lock} : \mathrm{Val} \to \mathrm{SProp}.$$
$$\{R\}\ \mathbf{new}\ \mathsf{Lock}()\ \{\mathsf{r}.\ \mathsf{lock}(\mathsf{r})\}$$
$$\{\mathsf{lock}(\mathsf{l})\}\ \mathsf{l.Acquire}()\ \{\mathsf{lock}(\mathsf{l}) * R\}$$
$$\{\mathsf{lock}(\mathsf{l}) * R\}\ \mathsf{l.Release}()\ \{\mathsf{lock}(\mathsf{l})\}$$
$$\wedge\ \ \forall x : \mathrm{Val}.\ \mathsf{lock}(x) \Leftrightarrow \mathsf{lock}(x) * \mathsf{lock}(x)$$

This specification introduces an explicit lock representation predicate, lock, which is freely duplicable.

We now formalise the proof sketch of the joins-based lock implementation from Section 2. Hence, for any predicate R, we have to define a concrete lock predicate and show that the above specifications for the lock operations hold for the concrete lock predicate.

Channel initialisation phase

$$\frac{}{\{\mathsf{emp}\}\,\mathbf{new}\ \mathsf{Join}()\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(\emptyset,\emptyset,\mathsf{r})\}}\ \textsc{Join}$$

$$\frac{}{\{\mathsf{join}_{\mathsf{ch}}(A,S,\mathsf{j})\}\,\mathbf{new}\ \mathsf{SyncChannel}(\mathsf{j})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(A,S\cup\{\mathsf{r}\},\mathsf{j})*\mathsf{ch}(\mathsf{r},\mathsf{j})*\mathsf{r}\notin A\cup S\}}\ \textsc{Sync}$$

$$\frac{}{\{\mathsf{join}_{\mathsf{ch}}(A,S,\mathsf{j})\}\,\mathbf{new}\ \mathsf{AsyncChannel}(\mathsf{j})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(A\cup\{\mathsf{r}\},S,\mathsf{j})*\mathsf{ch}(\mathsf{r},\mathsf{j})*\mathsf{r}\notin A\cup S\}}\ \textsc{Async}$$

Chord initialisation phase

$$\frac{}{\{\mathsf{join}_{\mathsf{pat}}(P,Q,\mathsf{j})*\mathsf{ch}(\mathsf{c},\mathsf{j})\}\mathsf{j}.\mathsf{When}(\mathsf{c})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{pat}}(P,Q,\mathsf{j})*\mathsf{pat}(\mathsf{r},\mathsf{j},\{\mathsf{c}\})\}}\ \textsc{When}$$

$$\frac{}{\left\{\begin{array}{r}\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\\ \mathsf{pat}(\mathsf{p},j,X)*\mathsf{ch}(\mathsf{c},j)\end{array}\right\}\mathsf{p}.\mathsf{And}(\mathsf{c})\left\{\begin{array}{r}\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\\ \mathsf{pat}(\mathsf{p},j,X\cup\{\mathsf{c}\})\end{array}\right\}}\ \textsc{And}$$

$$\frac{\forall Y\in\mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y)=X\Rightarrow \circledast_{y\in Y}P(y)\sqsubseteq\circledast_{y\in Y}Q(y)}{\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X)\}\,\mathsf{p}.\mathsf{Do}()\,\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\}}\ \textsc{Do1}$$

$$\frac{}{\left\{\begin{array}{l}\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X)*\circledast_{z\in Z}\mathsf{ch}(z,j)*\\ \quad\forall Y\in\mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y)=X\Rightarrow\\ \quad b\mapsto\left\{\begin{array}{l}\mathsf{join}_{\mathsf{call}}(P,Q,j)*\\ \circledast_{z\in Z}\mathsf{ch}(z,j)*\circledast_{y\in Y}P(y)\end{array}\right\}\left\{\begin{array}{l}\circledast_{y\in Y}Q(y)\\ \ \end{array}\right\}\end{array}\right\}\mathsf{p}.\mathsf{Do}(\mathsf{b})\,\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\}}\ \textsc{Do2}$$

Message phase

$$\frac{}{\{\mathsf{join}_{\mathsf{call}}(P,Q,j)*\mathsf{ch}(\mathsf{c},j)*P(a,\mathsf{c})\}\,\mathsf{c}.\mathsf{Call}()\,\{\mathsf{join}_{\mathsf{call}}(P,Q,j)*Q(a,\mathsf{c})\}}\ \textsc{Call}$$

Phase transitions

$$\frac{\forall c,a.\ c\in A\Rightarrow Q(a,c)=\mathsf{emp}}{\mathsf{join}_{\mathsf{ch}}(A,S,j)\sqsubseteq\mathsf{join}_{\mathsf{pat}}(P,Q,j)}\qquad\frac{}{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\sqsubseteq\mathsf{join}_{\mathsf{call}}(P,Q,j)}$$

$$\mathsf{ch}(c,j)\Leftrightarrow\mathsf{ch}(c,j)*\mathsf{ch}(c,j)\qquad\mathsf{join}_{\mathsf{call}}(P,Q,j)\Leftrightarrow\mathsf{join}_{\mathsf{call}}(P,Q,j)*\mathsf{join}_{\mathsf{call}}(P,Q,j)$$

Abstract predicates

$$\begin{array}{ll}\mathsf{pat} & :\tau_{\mathrm{pat}}\times\tau_{\mathrm{join}}\times\mathcal{P}_m(\tau_{\mathrm{chan}})\to\mathrm{SProp}\qquad\mathsf{ch}:\tau_{\mathrm{chan}}\times\tau_{\mathrm{join}}\to\mathrm{SProp}\\ \mathsf{join}_{\mathsf{ch}} & :\mathcal{P}_m(\tau_{\mathrm{chan}})\times\mathcal{P}_m(\tau_{\mathrm{chan}})\to\mathrm{SProp}\\ \mathsf{join}_{\mathsf{pat}},\mathsf{join}_{\mathsf{call}} & :(\mathcal{E}\to\mathrm{LProp})\times(\mathcal{E}\to\mathrm{LProp})\times\mathrm{Val}\to\mathrm{SProp}\end{array}$$

Here $\mathcal{P}_m(X)$ denotes the set of finite multi-subsets of X and

$$\tau_{\mathrm{join}}=\tau_{\mathrm{chan}}=\tau_{\mathrm{pat}}\stackrel{\mathrm{def}}{=}\mathrm{Val}\qquad\mathcal{E}\stackrel{\mathrm{def}}{=}\mathrm{Val}\times\tau_{\mathrm{chan}}$$
$$\pi_{\mathsf{ch}}(X)\stackrel{\mathrm{def}}{=}\{\pi_2(x)\mid x\in X\}:\mathcal{P}_m(\mathcal{E})\to\mathcal{P}_m(\tau_{\mathrm{chan}})$$

Fig. 5. Specification of the joins library

The channel pre- and postconditions do not change relative to the informal proof. For any pair of channels c_a and c_r we define the channel pre- and postcondition, $P(c_a, c_r), Q(c_a, c_r) : \mathcal{E} \to \text{LProp}$, as follows:

$$P(c_a,c_r)(a,c) = \begin{cases} \text{emp} & \text{if } c = c_a \\ R & \text{if } c = c_r \\ \bot & \text{otherwise} \end{cases} \qquad Q(c_a,c_r)(a,c) = \begin{cases} R & \text{if } c = c_a \\ \text{emp} & \text{if } c = c_r \\ \bot & \text{otherwise} \end{cases}$$

In the proof, c_a will be instantiated with the acquire channel and c_r with the release channel. Note that the logical argument a is simply ignored.

The lock predicate then asserts that there exists some join instance and that fields acq and rel refer to channels with the above channel pre- and postcondition.

$$\begin{aligned} \mathsf{lock}(x) = \exists a, r, j : \text{Val}.\ & a \neq r \wedge x.\mathsf{acq} \mapsto a * x.\mathsf{rel} \mapsto r \\ & * \ \mathsf{ch}(a, j) * \mathsf{ch}(r, j) * \mathsf{join}_{\mathsf{call}}(P(a, r), Q(a, r), j) \end{aligned}$$

We explicitly require that a and r are distinct to ensure that the above definition of P and Q by case analysis on the second argument is well-defined. The lock predicate only asserts partial ownership of fields acq and rel, to allow the lock predicate to be freely duplicated.

Below is a full proof outline for the lock constructor.

public Lock() {
  Join j; Pattern p;
{this.acq $\mapsto$ *null* $*$ this.rel $\mapsto$ *null* $* R$}
  j = **new** Join();
{this.acq $\mapsto$ *null* $*$ this.rel $\mapsto$ *null* $* R *$ $\mathsf{join}_{\mathsf{ch}}(\emptyset, \emptyset, \mathsf{j})$}
  acq = **new** SyncChannel(j);
  rel = **new** AsyncChannel(j);
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{ch}}(\{r\}, \{a\}, \mathsf{j}) * a \neq r * \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j})$}
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{pat}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r * \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j})$}
  p = j.When(acq).And(rel);
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r * a \neq r *$ $\mathsf{join}_{\mathsf{pat}}(P(a, r), Q(a, r), \mathsf{j})$
   $* \ \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j}) * \mathsf{pat}(\mathsf{p}, \mathsf{j}, \{a, r\})$}
  p.Do();
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{pat}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r * \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j})$}
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{call}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r * \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j})$}
  rel.Call();
{this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{call}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r * \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j})$}
{lock(this)}
}

The call to Do further requires that we prove:

$$\forall Y \in \mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y) = \{a, r\} \Rightarrow \circledast_{y \in Y} P(a, r)(y) \sqsubseteq \circledast_{y \in Y} Q(a, r)(y)$$

which follows easily from the commutativity of $*$.

The full proof outline for Acquire is given below. The proof for Release is similar.

public void Acquire() {
  SyncChannel c;

$\{\mathsf{lock(this)}\}$
$\{\mathsf{this.acq} \mapsto\!\!\!\!\sim\; a * \mathsf{this.rel} \mapsto\!\!\!\!\sim\; r * \mathsf{join}_{\mathsf{call}}(P(a,r), Q(a,r), \mathsf{j}) * a \neq r * \mathsf{ch}(a,\mathsf{j}) * \mathsf{ch}(r,\mathsf{j})\}$
  $\mathsf{c} = \textbf{this}.\mathsf{acq};$
$\{\mathsf{this.acq} \mapsto\!\!\!\!\sim\; \mathsf{c} * \mathsf{this.rel} \mapsto\!\!\!\!\sim\; r * \mathsf{join}_{\mathsf{call}}(P(\mathsf{c},r), Q(\mathsf{c},r), \mathsf{j}) * \mathsf{c} \neq r * \mathsf{ch}(\mathsf{c},\mathsf{j}) * \mathsf{ch}(r,\mathsf{j})\}$
  $\mathsf{c.Call}();$
$\{\mathsf{this.acq} \mapsto\!\!\!\!\sim\; \mathsf{c} * \mathsf{this.rel} \mapsto\!\!\!\!\sim\; r * \mathsf{join}_{\mathsf{call}}(P(\mathsf{c},r), Q(\mathsf{c},r), \mathsf{j}) * \mathsf{c} \neq r$
    $* \mathsf{ch}(\mathsf{c},\mathsf{j}) * \mathsf{ch}(r,\mathsf{j}) * Q(\mathsf{c},r)(0,\mathsf{c})\}$
$\{\mathsf{lock(this)} * R\}$
}

When we call the acq channel we have to pick a logical argument a. Since the channel pre- and postcondition ignores the a, we can pick anything. In the above proof we arbitrarily picked 0, hence the $Q(\mathsf{c},r)(0,\mathsf{c})$ in the postcondition.

5.2 n-Barrier

In this section we formalise a proof of the n-barrier from Section 2.4. This example illustrates how logical arguments combined with phantom state allows us to logically distinguish messages on a single channel. The example also illustrates the use of a non-trivial view-shift to update a phantom field upon firing of a chord.

Desired Specification. In Section 2.4 we gave an informal specification of an n-barrier, under the assumption that clients transferred the same resources to the barrier at every round of synchronisation, and that the barrier redistributed these resources in the same way at every round of synchronisation. As these assumptions are unrealistic, we start by generalising the specification.

The simplified n-barrier specification was expressed in terms of two assertions B_i^{in} and B_i^{out} that described the resources client i transferred to and from the barrier at every round of synchronisation. Here, instead, we take B^{in} and B^{out} to be predicates indexed by a client identifier i and the current round of synchronisation m. The general n-barrier specification is given in Figure 6.

$$
\begin{array}{l}
\forall n \in \mathbb{N}.\ \forall B^{\mathsf{in}}, B^{\mathsf{out}} : \{1,...,n\} \times \mathbb{N} \to \mathrm{LProp}. \\
\quad (\forall m \in \mathbb{N}.\ \circledast_{i\in\{1,...,n\}} B_i^{\mathsf{in}}(m) \sqsubseteq \circledast_{i\in\{1,...,n\}} B_i^{\mathsf{out}}(m))) \ \Rightarrow \\
\qquad \exists \mathsf{barrier} : \mathrm{Val} \to \mathrm{SProp}.\ \exists \mathsf{client} : \mathrm{Val} \times \{1,...,n\} \times \mathbb{N} \to \mathrm{SProp}. \\
\qquad\quad \{\mathsf{n} = n\}\, \textbf{new}\ \mathsf{Barrier(n)}\, \{\mathsf{ret}.\ \mathsf{barrier(ret)} * \circledast_{i\in\{1,...,n\}} \mathsf{client(ret},i,0)\} \\
\qquad\quad \wedge\ \forall i \in \{1,...,n\}.\ \forall m \in \mathbb{N}. \\
\qquad\qquad \{\mathsf{barrier(b)} * \mathsf{client(b},i,m) * B_i^{\mathsf{in}}(m)\} \\
\qquad\qquad\quad \mathsf{b.Arrive()} \\
\qquad\qquad \{\mathsf{barrier(b)} * \mathsf{client(b},i,m+1) * B_i^{\mathsf{out}}(m)\} \\
\qquad\quad \wedge\ \forall x : \mathrm{Val}.\ \mathsf{barrier}(x) \Leftrightarrow \mathsf{barrier}(x) * \mathsf{barrier}(x)
\end{array}
$$

Fig. 6. General n-barrier specification. This specification requires that the number of clients, n, is known statically. This simplifies the exposition. We can also specify and verify a specification without this assumption.

Here barrier is the barrier representation predicate, which can be freely duplicated. The client predicate plays two roles: namely, (1) to witness the identity of each barrier client (like the id predicate from Section 2.4), and (2) to ensure that every client of the barrier agrees on the round of synchronisation, m, whenever they arrive at the barrier. These two properties are necessary to ensure that we can redistribute the combined resources when every client has arrived at the barrier. When one creates a new n-barrier, one thus receives n client predicates – one for each client – each with 0 as the current round of synchronisation. The current round of synchronisation is incremented by one at each arrival at the barrier.

Predicate Definitions. We start by giving concrete definitions for the abstract barrier and client predicate. Hence, assume $n \in \mathbb{N}$ clients and abstract predicates $B^{\mathsf{in}}, B^{\mathsf{out}} : \{1, ..., n\} \times \mathbb{N} \to \mathrm{LProp}$ satisfying,

$$\forall m \in \mathbb{N}.\ \circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{in}}(m) \sqsubseteq \circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{out}}(m) \tag{2}$$

Since the n-barrier only has a single channel, we need to pick a single channel pre- and postcondition that works for every client, for every round of synchronisation. We thus take the logical argument for the arrival channel to be a pair consisting of a client identifier i and the current synchronisation round m. From the specification above, when client i arrives for synchronisation round m it transfers $B_i^{\mathsf{in}}(m)$ to the barrier and expects to receive back $B_i^{\mathsf{out}}(m)$. In addition, the client gives up its client predicate and gets back a new one, with the same logical client identifier i and an incremented synchronisation round, $m + 1$. For any barrier b and channel c_1 we thus define the channel pre- and postcondition $P(b, c_1), Q(b, c_1) : \mathcal{E} \to \mathrm{LProp}$ as follows:

$$P(b,c_1)((i,m),c) = \begin{cases} \mathsf{client}(b,i,m) * B_i^{\mathsf{in}}(m) & \text{if } c = c_1 \\ \bot & \text{otherwise} \end{cases}$$

$$Q(b,c_1)((i,m),c) = \begin{cases} \mathsf{client}(b,i,m+1) * B_i^{\mathsf{out}}(m) & \text{if } c = c_1 \\ \bot & \text{otherwise} \end{cases}$$

Here the (i, m) is the logical argument consisting of the logical client identifier i and synchronisation round m. In the proof, c_1 will be instantiated with the arrival channel.

Above, we defined the channel pre- and postcondition in terms of an abstract client predicate, which we have not defined yet. We thus need to define client. This is the main technical challenge of the proof. So, to motivate its definition, we start by considering what properties the client predicate should satisfy. Recall that we use the client predicate to (1) witness the identity of clients, and to (2) ensure that clients agree on the current round of synchronization when they arrive at the barrier.

To witness the identity of clients, disjoint client predicates must refer to distinct clients, as expressed by property (3) below. To ensure that clients agree on the current round of synchronisation, the client predicate should also satisfy (4). Lastly, to update the current round of synchronisation when every client has arrived at the barrier, the client predicate should satisfy (5).

$$\forall b, i, j, m.\ \mathsf{client}(b, i, m) * \mathsf{client}(b, j, m) \Rightarrow i \neq j \tag{3}$$

$$\forall b, i, j, m, k.\ \mathsf{client}(b, i, m) * \mathsf{client}(b, j, k) \Rightarrow m = k \tag{4}$$

$$\forall b, m.\ \circledast_{i \in \{1,\ldots,n\}} \mathsf{client}(b, i, m) \sqsubseteq \circledast_{i \in \{1,\ldots,n\}} \mathsf{client}(b, i, m+1) \tag{5}$$

Note that (5) is consistent with (4), since we update all n client predicates simultaneously.

We can satisfy (4) and (5) by introducing a phantom field to keep track of the current round of synchronisation. By giving each client $1/n$-th permission of this phantom field, we ensure that every client agrees on the current round of synchronisation, (4). Furthermore, given all n client predicates, these fractions combine to the full permission, allowing the phantom field to be updated arbitrarily, and thus in particular, to be incremented; thus satisfying (5). We can satisfy (3) by associating each client identifier i with a non-duplicable resource $\bullet_i$ in the logic, and requiring ownership of $\bullet_i$ in the client predicate. We thus define client as follows, $\mathsf{client}(b, i, m) = b_{\mathsf{round}} \overset{1/n}{\longmapsto} m * \bullet_i^b$, where $\bullet_i^b$ is defined as follows: $\bullet_i^b = \exists v : \mathrm{Val}.\ b_i \mapsto v$.

The barrier predicate is now trivial to define:

$$\mathsf{barrier}(b) = \exists j, c : \mathrm{Val}.\ b.\mathsf{arrive} \overset{-}{\mapsto} c * \mathsf{join}_{\mathsf{call}}(P(b, c), Q(b, c), j) * \mathsf{ch}(c, j)$$

It simply asserts that arrive refers to a channel on a join instance with the channel pre- and postcondition we defined above.

Proof. Now that we have defined a client predicate satisfying (3), (4), and (5), we can proceed with the verification of the n-barrier. The main proof obligation is proving that the barrier chord satisfies the postconditions of the channels it matches. Since the barrier chord matches n arrival messages, by rule Do1 we thus have to prove that:

$$\forall Y \in \mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y) = \{c_1^n\} \Rightarrow \circledast_{y \in Y} P(b, c_1)(y) \sqsubseteq \circledast_{y \in Y} Q(b, c_1)(y)$$

To simplify the exposition, we consider the case for $n = 2$. The proof for $n > 2$ follows the same structure. For $n = 2$ the above proof obligation reduces to:

$$\begin{aligned} &\forall i_1, i_2, m_1, m_2. \\ &\quad \mathsf{client}(b, i_1, m_1) * B_{i_1}^{\mathsf{in}}(m_1) * \mathsf{client}(b, i_2, m_2) * B_{i_2}^{\mathsf{in}}(m_2) \ \sqsubseteq \\ &\qquad \mathsf{client}(b, i_1, m_1 + 1) * B_{i_1}^{\mathsf{out}}(m_1) * \mathsf{client}(b, i_2, m_2 + 1) * B_{i_2}^{\mathsf{out}}(m_2) \end{aligned} \tag{6}$$

At this point we cannot directly apply the user-supplied redistribution property, (2), as it requires that $m_1 = m_2$ and $i_1 \neq i_2$. First, we need to use properties (3) and (4) to constrain what logical arguments clients could have chosen when they send their arrival messages. By property (4) it follows that $m_1 = m_2$. Furthermore, from property (3) it follows that i_1 and i_2 are distinct. Since $i_1, i_2 \in \{1, 2\}$, (6) thus reduces to:

$$\begin{aligned} &\forall m.\ \mathsf{client}(b, 1, m) * B_1^{\mathsf{in}}(m) * \mathsf{client}(b, 2, m) * B_2^{\mathsf{in}}(m) \ \sqsubseteq \\ &\qquad \mathsf{client}(b, 1, m + 1) * B_1^{\mathsf{out}}(m) * \mathsf{client}(b, 2, m + 1) * B_2^{\mathsf{out}}(m) \end{aligned} \tag{7}$$

Using the redistribution property, (2), and (5) it follows that,

$$\forall m.\ B_1^{\mathsf{in}}(m) * B_2^{\mathsf{in}}(m) \sqsubseteq B_1^{\mathsf{out}}(m) * B_2^{\mathsf{out}}(m)$$
$$\forall m.\ \mathsf{client}(b,1,m) * \mathsf{client}(b,2,m) \sqsubseteq \mathsf{client}(b,1,m+1) * \mathsf{client}(b,2,m+1)$$

Combining these two we thus get (7). We have thus proven (6). Note that here we implicitly used the ability to perform a view shift when a chord fires, to increment the value of the phantom field round.

The verification of the constructor and Arrive method is now straightforward.

In summary, using logical arguments and phantom state we can thus show that the generalised n-barrier from Section 2.4 satisfies the generalised barrier specification. While the proof is more technically challenging than any of the previous examples, it is still a high-level proof about *barrier* concepts. Informally, we proved that clients agree on the current synchronisation round and that clients identify themselves correctly; both natural proof obligations for a barrier.

6 Discussion

We first relate our specification of joins and the clients thereof to earlier work and then evaluate what we have learned about HOCAP from this case study.

In terms of reasoning about external sharing, O'Hearn's original concurrent separation logic supports reasoning about shared variable concurrency using critical regions [17]. This was subsequently extended to a language with locking primitives by Hobor et al. [11] and Gotsman et al. [10], and to a language with barrier primitives by Hobor et al. [12]. In all four cases, the underlying synchronisation primitives were taken as language primitives and their soundness was proven meta-theoretically.

Concurrent abstract predicates by Dinsdale-Young et al. [7] extends standard separation logic with support for reasoning about shared mutable state by imposing protocols on shared resources. Dinsdale-Young et al. used this logic to verify a spin-lock implemented using compare-and-swap. The spin-lock was verified against a non-standard lock specification *without* built-in support for reasoning about external sharing. Hence, to reason about external sharing, *clients* would have to define a protocol of their own, relating ownership of the shared resources with the state of the lock. This type of reasoning is *not* modular, as it requires the specification of concurrent libraries to expose *internal* implementation details of synchronisation primitives, to allow clients to define a protocol governing the external sharing.

Jacobs and Piessens recently extended their VeriFast tool with support for fine-grained concurrency [14] and verified a lock-based barrier implementation [13] inspired by [11]. They verify the implementation against a specification without built-in support for reasoning about external sharing. Compared to our barrier specification, their specification is thus fairly low-level, requiring *clients* of the barrier to use auxiliary variables to encode who has arrived and what resources they have transferred to the barrier.

The goal of this case study was to test whether HOCAP supports modular reasoning about concurrent higher-order imperative libraries. To this end, we have

proposed an abstract specification of the C♯ joins library, expressed in terms of high-level join primitives. We have demonstrated that this abstract specification suffices for formal reasoning about a series of classic synchronisation primitives, which allow for external sharing. Compared to previous work on verifying synchronisation primitives using separation logic, our specifications are stronger and our proofs are considerably simpler. Thus, from this perspective, our case study supports the thesis that HOCAP is useful for modular reasoning about concurrent higher-order imperative libraries. However, as explained in Section 3, the joins specification presented in this paper is restricted to local pre- and postconditions for channels, which means that synchronization primitives implemented using joins can only have local assertions as resource invariants. Recall, e.g., the lock specification in Section 5.1, where the resource invariant ranges over LProp, which means that clients of the lock cannot use CAP when picking a resource invariant for the lock. In the technical report [24] we have presented a stronger specification of joins, which does allow clients to use CAP for such resource invariants, but that is at the expense of complicating the specification, to avoid circular sharing patterns. Thus future work includes finding stronger models of HOCAP that support simple specifications and circular sharing patterns.

Acknowledgements. We would like to thank Mike Dodds, Bart Jacobs, Jonas Braband Jensen, Hannes Mehnert, Claudio Russo, and Aaron Turon for helpful discussions and feedback.

References

1. Biering, B., Birkedal, L., Torp-Smith, N.: BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. ACM TOPLAS (2007)
2. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: Proceedings of POPL, pp. 259–270 (2005)
3. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
4. Courtois, P.-J., Heymans, F., Parnas, D.L.: Concurrent Control with "Readers" and "Writers". Commun. ACM 14(10), 667–668 (1971)
5. da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., Wheelhouse, M.: A simple abstraction for complex concurrent indexes. SIGPLAN Not. 46(10), 845–864 (2011)
6. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: Compositional Reasoning for Concurrent Programs. In: Proceedings of POPL (2013)
7. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
8. Fournet, C., Gonthier, G.: The reflexive chemical abstract machine and the join-calculus. In: Proceedings of POPL (1996)
9. Fournet, C., Gonthier, G.: The Join Calculus: A Language for Distributed Mobile Programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)

10. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local Reasoning for Storable Locks and Threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)
11. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle Semantics for Concurrent Separation Logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
12. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 276–296. Springer, Heidelberg (2011)
13. Jacobs, B.: Verified general barriers implementation, `http://people.cs.kuleuven.be/ bart.jacobs/verifast/examples/barrier.c.html`
14. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Proceedings of POPL, pp. 271–282 (2011)
15. Krishnaswami, N.: Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic. PhD thesis, Carnegie Mellon University (2012)
16. Krishnaswami, N., Birkedal, L., Aldrich, J.: Verifying Event-Driven Programs using Ramified Frame Properties. In: Proceedings of TLDI (2010)
17. O'Hearn, P.W.: Resources, Concurrency and Local Reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
18. Owicki, S.S.: Axiomatic Proof Techniques for Parallel Programs. PhD thesis, Cornell (1975)
19. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Proceedings of POPL, pp. 247–258 (2005)
20. Russo, C.V.: The Joins Concurrency Library. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 260–274. Springer, Heidelberg (2007)
21. Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested Hoare Triples and Frame Rules for Higher-Order Store. LMCS, 7(3:21) (2011)
22. Svendsen, K., Birkedal, L., Parkinson, M.: Verifying Generics and Delegates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 175–199. Springer, Heidelberg (2010)
23. Svendsen, K., Birkedal, L., Parkinson, M.: Higher-order Concurrent Abstract Predicates. Technical report, IT University of Copenhagen (2012), `http://www.itu.dk/people/kasv/hocap-tr.pdf`
24. Svendsen, K., Birkedal, L., Parkinson, M.: Verification of the Joins Library in Higher-order Separation Logic. Technical report, IT University of Copenhagen (2012), `http://www.itu.dk/people/kasv/joins-tr.pdf`
25. Svendsen, K., Birkedal, L., Parkinson, M.: Impredicative Concurrent Abstract Predicates. Under submission (2013)
26. Svendsen, K., Birkedal, L., Parkinson, M.: Modular Reasoning about Separation of Concurrent Data Structures. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 169–188. Springer, Heidelberg (2013)

Enabling Modularity and Re-use in Dynamic Program Analysis Tools for the Java Virtual Machine

Danilo Ansaloni[1], Stephen Kell[1], Yudi Zheng[1], Lubomír Bulej[2], Walter Binder[1], and Petr Tůma[2]

[1] University of Lugano, Switzerland
firstname.lastname@usi.ch
[2] Charles University, Prague, Czech Republic
firstname.lastname@d3s.mff.cuni.cz

Abstract. Dynamic program analysis tools based on code instrumentation serve many important software engineering tasks such as profiling, debugging, testing, program comprehension, and reverse engineering. Unfortunately, constructing new analysis tools is unduly difficult, because existing frameworks offer little or no support to the programmer beyond the incidental task of instrumentation. We observe that existing dynamic analysis tools re-address recurring requirements in their essential task: maintaining state which captures some property of the analysed program. This paper presents a general architecture for dynamic program analysis tools which treats the maintenance of analysis state in a modular fashion, consisting of *mappers* decomposing input events spatially, and *updaters* aggregating them over time. We show that this architecture captures the requirements of a wide variety of existing analysis tools.

1 Introduction

To gain insight about how to optimise, debug, extend and refactor large systems, programmers often rely on dynamic program analysis tools, which observe a program in execution and report properties of that execution. Many popular bug-finding and profiling tools are of this form, including the Valgrind suite [25], DTrace [5], VisualVM[1], GProf [13] and others, while research literature continues to propose diverse new tools for data race detection [11], white-box testing [29], security policy enforcement [10] and more.

Implementing such tools is unduly difficult. One recurring source of difficulty is that high-level requirements must be translated into code reacting to low-level execution events. For example, a simple context-sensitive memory profiler, which must "count allocated bytes, totalled by call chain" must be written in terms of method entries and exits, a variety of low-level object allocation mechanisms, and so on. Although several existing frameworks assist with creation of dynamic analyses, all support only the same broad approach, which we characterize as

[1] http://visualvm.java.net/

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 352–377, 2013.

control flow interposition: they provide the interception mechanisms for various control-flow events, but it remains the user's responsibility to describe how their analysis abstracts and aggregates these events. This approach is found in bytecode transformation libraries such as ASM[2], Soot[3] or Javassist [6], in aspect languages such as AspectJ [18], in external domain-specific languages (DSLs) such as DTrace's D [5][4] and in embedded DSLs such as BTrace[5] or DiSL [23]. By abstracting only the control-flow aspect of the task, considerable work is repeated by successive tool authors in bridging the abstraction gap between high-level tool requirements and low-level instrumentation.

In this paper we present a system for describing dynamic analyses more succinctly using a contrasting approach which we describe as *state-oriented*. The analysis' requirements are decomposed in terms of the structures which hold the accumulated state of the analysis, and the semantics with which these structures evolve. For example, our allocation profiler is defined as a composition of a call chain recorder, a map from call chains to counts, an encapsulated definition of allocation events, and an updater function which increments the relevant count on each allocation. All these are re-usable library components. In our system, dynamic analyses are built straightforwardly by using short script-style code fragments to combine generic data structures and state transformers.

Our goal is that using such a system, a large proportion of analysis requirements can be catered for almost entirely by library code. In other cases, new requirements can easily be satisfied by creating small extensions of existing library code. In rare cases where library code cannot be combined or extended to satisfy requirements directly, the full expressiveness of control-flow interposition is available, since our system builds on an existing, more conventional framework. So, while we preserve the expressiveness of existing control-oriented systems, common cases are handled using library code rather than new user code.

The contributions of this paper are threefold.

- We describe the design of a framework for composition of dynamic analyses, focusing on its API and three key constituent interfaces which facilitate our state-oriented decomposition: *instrumentations* emitting events when relevant code regions are being executed; *updaters* describing how the analysis's state (meaning the state which is used to produce the analysis's output, such as profile data or execution monitor state) responds to new events; and *mappers* routing events to the subset of output state requiring consequent update.
- We show with examples that this factoring can express a wide variety of analyses, including substantial real use cases presented as case studies.

[2] `http://asm.ow2.org/`

[3] `http://www.sable.mcgill.ca/soot`

[4] We note that control-flow abstractions are a characteristic of common DTrace providers, but not necessarily of the whole framework.

[5] `http://kenai.com/projects/btrace`

– We show experimentally that our system offers performance generally competitive with lower-level frameworks used like-for-like.

We begin by motivating the high-level design of our system.

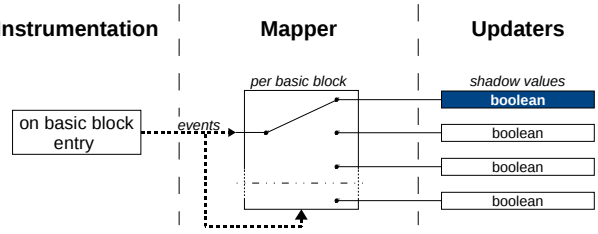
2 Motivation

In this section we motivate our work by showing that (1) significant latent commonality exists among dynamic analysis tools, (2) frameworks currently used to build them do not extract this commonality, and (3) a more state-oriented decomposition of analyses can ameliorate this. We consider these issues in turn.

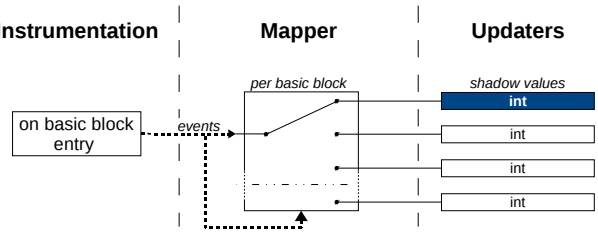
2.1 Latent Commonality

Instrumentation is only one of the recurring sources of complexity in dynamic analysis tools. The state maintained by dynamic analysis tools is commonly explained using the concept of *shadowing*: the job of the analysis is to update a set of shadow values that correspond to program state elements such as objects, fields or threads. The shadows' role is to maintain additional data about the program's execution so far, supplementing the program state. Each shadow value is updated in response to incoming events. Update rules might simply involve incrementing an integer (in the case of counters maintained by a profiler), or might manipulate a set (in the case of a data-race detector based on lock sets), propagate taint bits (in the case of an information flow analysis), advance a finite state automaton, and so on. Our insight is that by separating out the programmer's description of *how* shadows are represented and updated from *what* unit of program state is modelled by each shadow, greater commonality can be extracted re-usably from distinct tools.

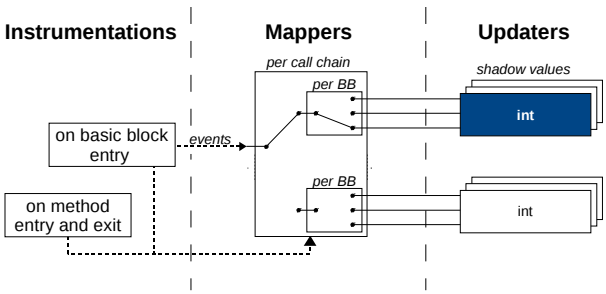
To illustrate this, we observe how a variety of useful tools can be constructed by independently recombining different shadow value mappings (i.e., what elements of program state to shadow) with different update rules (i.e., how to shadow them). First, consider a simple code coverage tool working at the basic block level. It consists of (1) an instrumentation of basic block entries, (2) a mapper associating a shadow boolean value to every distinct basic block ID, and (3) an updater that, for each basic block ID that is received, sets its shadow boolean to true.



Instead of coverage, suppose we now require a count-based profiling tool. We shadow the same program state elements, but now with an integer updated by increments (instead of a boolean updated by set-to-true).



Now consider a context-sensitive profiler. We keep the same updater, but maintain each shadow per call chain. This means our mapper now has two levels: from call chain, to basic block, to the counter payload. The set of call chains must itself be constructed by additional instrumentation, applied to method entry and exit, typically to maintain a calling context tree [1].



Note that the overall form of the system is still the same, and it contains the same kinds of units: an instrumentation that observes events of interest, a mapper of such events to the relevant part of the analysis state (possibly over multiple stages), and updaters of individual state elements in response to events to reflect the context information available for such events. An interesting property is that the mapping logic can itself maintain state and be sensitive to events gathered using instrumentation, as with the call chain in the latter example.

To move from each example to the next in the above series, we change either how each analysis state element (shadow) is represented and updated (booleans updated by set-to-true, versus integers updated by increment), or what spatial structure of shadow elements the gathered events are mapped onto, including how fine-grained this is (from "one shadow per basic block" to "one per basic block, per call chain"). However, in no case we change both at the same time. Therefore, if it were possible for the analysis developer to specify all these concerns independently, each analysis could be constructed very simply by re-using pieces of the previous one. Achieving this independence is a key contribution of our API design (§3.4).

2.2 Limitations of Current Infrastructure

Fig. 1 summarises our goals: to provide still greater ease of use without sacrificing flexibility. To achieve ease of use, we seek greater re-use in the code used to construct dynamic analysis tools. Our insight is that despite the latent commonality we saw in the previous section, current infrastructure makes it either impossible or extremely difficult to structure dynamic analysis tools so that this logic can be isolated and re-used. We survey these existing infrastructures in two broad categories: low-level libraries for code transformation, and higher-level instrumentation-based frameworks.

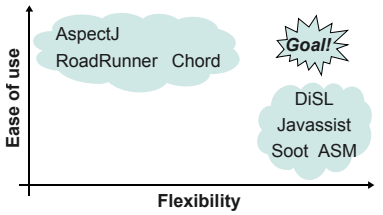


Fig. 1. Trades of ease-of-use with flexibility in existing frameworks

Low-level libraries A basic yet popular infrastructure supporting dynamic analyses is libraries for manipulating code representations such as Java bytecode [20], VEX [25], LLVM [19], or machine instructions. Convenient APIs for transformation of this representation include libVEX[6], ASM[7], Soot and Javassist [6]. These reduce the effort required to perform common-case transformations, relative to manual coding. However, they do not abstract away the complexity of this representation, since the programmer must still have a thorough understanding of it in order to use the API.[8]

Fixed-event-set frameworks Many analysis frameworks focus on relieving the programmer of the instrumentation burden, by abstracting various low-level instrumentations into the form of a fixed set of hooks (to which callbacks may be registered) or join points (for which advice may be provided) or events (for which handlers may be defined). These frameworks are valuable, but their API designs limit their ability to construct analyses by composition, as we now detail.

Unicast events In Chord[9], for example, an analysis is defined as an implementation of the EventHandler abstract class, implementing a fixed set of

[6] `http://www.valgrind.org/`
[7] http://asm.ow2.org/
[8] We note that these systems are powerful, and are not limited to the construction of analysis tools; they can replace or transform entire sections of code in arbitrary ways. Here, we are referring only to their suitability with respect to construction of program analysis tools (a common use case).
[9] `http://pag.gatech.edu/chord/`

callback-style methods. Since there may be only one EventHandler instance deployed by a given analysis, there is no convenient way to combine independent analyses reacting to the same or overlapping sets of events (without manual forwarding code). The pipeline architecture of RoadRunner [12] also suffers from a unicast limitation, in that once an event has been discarded by an earlier pipeline stage, it is unavailable to all later stages.

Event forwarding requirements Since Chord targets Java, where only single inheritance is provided, it cannot combine independently-written handlers for disjoint sets of events without a specially crafted EventHandler that forwards events as required.

Limitations of pipeline-style composition Although RoadRunner provides an apparently modular pipeline-based composition model, each pipeline stage makes strong assumptions about the shadowing and forwarding behaviour of the rest of the pipeline.[10] As a result, although pipeline stages are replaceable by algorithmically distinct alternatives (a primary design goal of the framework [11]) they are not highly re-usable, since each one is usable only in a pipeline contexts similar to the one for which it was designed.

Fragile base classes In both Chord and RoadRunner, defining a new kind of event creates fragile base class problems: in Chord, it requires defining a new Instrumentation class (using the lower-level Javassist library) dispatching to a new EventHandler-like base class. Existing analyses cannot be rebased onto this new superclass without forwarding code. RoadRunner has a similar property: since events are propagated through distinct methods in the Tool base class, defining new events means modifying this base class.

To avoid the problems just seen, we must design an API that does not model event transmission as delegation within an inheritance hierarchy, nor as method calls across a fixed interface.

2.3 State-Oriented Abstractions

Besides an appropriate event abstraction, we must provide additional abstractions to capture the commonality seen in §2.1. Invariably, as we saw with the examples of §2.1, non-trivial dynamic analyses maintain some kind of state; describing this state can offer a succinct way to describe the analysis.

We claim that any instrumentation-based dynamic analysis may be decomposed roughly using the following equation.

$$Analysis = Instrumentations + Mappers + Updaters$$

Instrumentations provide events from the base program. *Updaters* modify the "primary" state of the analysis, meaning the information whose collection is the end goal of the analysis. This might include the relative hotness of each

[10] We note that RoadRunner's implementation of shadow state allows each field in an object to be shadowed, but by at most one pipeline stage at a given time. This forces pipeline stages to coordinate hand-off, using a shared state machine.

method or basic block (for a CPU profiler) or the state of each active execution monitor (for a bug-finding analysis). Updaters include the logic which updates these values in response to new events.[11] Finally, *mappers* connect the other two layers; logically speaking, they route the incoming events to a subset of the relevant shadow values.

A key insight of our design is that the mapping stage is often an independently re-usable part, or, in complex cases, a composition of such parts. For example, in §2.1 we saw context-sensitive data collection implemented using a calling context tree. Such a tree represents an intermediate mapping layer which is independently re-usable across many analyses. In general, the same update logic might be valid for a variety of granularities of analysis—depending, for example, on whether profiles are being collected per method or per call chain, whether monitoring is done per-object or per-class, and so on. A separate mapping stage allows these granularities to be defined independently from other aspects of the analysis.

Table 1 offers some credence to the claim of our equation's generality. It shows several analyses, including both simple examples of the kind seen earlier (§2.1) and complex examples from the literature, decomposed according to this equation. The table columns refer to the concepts we have by now introduced. The "cardinality" column elaborates on the role of the mapping stage: most mapping stages are designed to maintain one shadow value per some division of program state (current or former), such as "one state machine per live object" or "one count per call chain seen", and we detail this explicitly.

To realise this decomposition, we require a programmatic way to express each of the three parts separately. Furthermore, our programmatic realisation must be efficient so as to meet the requirements of dynamic program analysis. In the next section, we therefore describe this programmatic realisation in the form of an API design, paying special attention to both the overall ease-of-use and the essential implementation details on which the design's practicability depends.

3 API Design

In this section we present FRANC, our FRamework for ANalysis Composition, which is implemented on top of the DiSL instrumentation framework[12] [23], and which models our state-oriented decomposition of dynamic analysis tools. Like DiSL, FRANC targets Java bytecode. We chose DiSL partly because it is an open-source framework and (to our knowledge) the only one providing full bytecode coverage out-of-the-box. Whereas DiSL provides instrumentation primitives and a set of annotations allowing Java-language code snippets (static methods) to be inlined in a user-specified manner, FRANC instead focuses on encapsulating instrumentation behind "listener"-like[13] event interfaces, and subsequently processing events in plain, annotation-free Java code. Instrumentations are event

[11] We will sometimes prefer to talk about collective "analysis state", and sometimes about many individual "shadow values".

[12] `http://disl.ow2.org/`

[13] We use "listener" in the same sense as various Java libraries, notably Swing [33].

name	instrumentations	mappers		updaters	
		function of	cardinality	shadow values	update rule
basic block coverage	basic block entry	event only	per basic block	boolean	set true on entry
basic block hotness	basic block entry	event only	per basic block	counter	increment on entry
context-sensitive basic block profiler	basic block entry	calling context tree	per call chain × basic block	counter	increment on entry
context-sensitive allocation profiler	memory allocation	calling context tree	per allocation call chain	counter	increment (by alloc size) on entry
cache simulator (e.g. [32])	memory read/write	accessed line + cache set state	per access site × $\{hit, miss\}$	counter	increment on hit/miss
ElephantTracks [26]	method entry/exit object allocation object use object reclamation reference update	event + timestamp (using logical clock)	per object + single value (logical clock)	object last-use time referenced objects	Merlin algorithm [14]
listener latency profiling [16]	method entry/exit	event + timestamp (using system clock)	per subtype of EventListener	($Count$, $AverageDuration$)	running average update
typestate checker [31]	method entry	event only	per live object	typestate automaton	advance, else report violation
concolic execution [29]	value-generating computation, branches	path constraint	per program value	SMT formulae	update + simplify
Eraser race detector [28]	lock ops, field read/write	event only	per pair ($object$, $field$)	consistent lock set	initialize, then set-intersect; report if empty
dynamic shape analysis [17]	garbage collector events: object scan complete, heap traversal complete	event only	per class	class-field summary graph	initialize, then invariance check
DefUse [30]	field read/write	event only	per field	writing thread ID or instruction	invariance check

Table 1. A series of distinct dynamic analyses decomposed into events, mappers and shadow values

producers. The remainder of the analysis is implemented as event consumers which subscribe to some set of instrumentations. The underlying DiSL API is available to handle rare cases where library-defined instrumentations do not suffice. In overview, our API provides the following features:

- library-defined event sources, implemented by bytecode instrumentation;
- mappers that aggregate, decompose, augment and/or filter incoming events—and may be built compositionally from other mappers;
- updaters that comprise a representation for an element of analysis output state and logic for updating it in response to events;
- "auto-completion" features for inferring common wiring patterns between event producers and consumers.

Clients of the FRANC API are submitted to a weaver which analyses the client program's use of various API interfaces and performs the necessary instrumentation. Our use of a Java API reflects our implementation, which is both a Java-based API and performs instrumentation of Java bytecode. However, we believe the same conceptual decomposition could be implemented conveniently in many other instrumentation scenarios.

3.1 API Elements

Each element of our equation (§2.3) corresponds to a small set of API-provided definitions. We briefly survey these now, before proceeding to a full example.

Instrumentations We can construct instrumentations using code like the following. FieldAccess is one of a collection of *code region* marker interfaces identifying various bytecode patterns which can be instrumented.

```
Instrumentation<FieldAccess> faInstr = new Instrumentation<>();
```

Scheduling interfaces Code which consumes instrumentation events implements another kind of marker interface, called scheduling interfaces, that decide whether the code executes *Before* or *After* the instrumented feature (among other options). Here we begin a class that will define code to run after each field access.

```
class FieldMapper
  implements After<FieldAccess>
  // ... to be continued
```

Shadow value maps The shadow state of our analysis is usually stored in some object implementing Java's standard Map interface. Here we store a set of counters, indexed by strings (whose values will be explained shortly).

```
Map<String, AtomicLong> fieldAccesses = new HashMap<>();
```

Mappers Mappers are a particular kind of client of instrumentation. Usually, they use instrumentation events to maintain a current value—on either a global, per-thread, or some other basis. The current value denotes the key (in the shadow state map) that is currently relevant for update. For example, our field access mapper holds a string representing the most recently accessed field. We represent these current values using familiar Java data types (modestly generalised) such as ThreadLocal. We have already seen the beginning of the mapper definition; in full, it is as follows.

```
class FieldMapper
  extends ThreadLocal<String>
  implements After<FieldAccess> {
    public void after(FieldAccess codeRegion)
    { set(FieldAccessContext.getFullFieldName(codeRegion)); } // grab current field
}
FieldMapper currentField = new FieldMapper();
```

Context accessors The helper class FieldAccessContext, which retrieves a unique string identifying the field, belongs to a library of context accessor classes, whose design confers important modularity properties. We return to this in §3.3.

Updaters How does a new FieldAccess event update the relevant element of the fieldAccesses map? This is encapsulated in an updater object. Ours comes from the library, and has simple "increment" semantics.

```
Analysis<FieldAccess> updater = new PostIncrement<>(fieldAccesses, currentField);
```

Subscription relationships To "wire up" the system, we can explicitly create subscription relationships to route events from an Instrumentation to its clients—here both the mapper and the updater.

```
faInstr.appendSubscriber(currentField);
faInstr.appendSubscriber(updater);
```

3.2 Complete Example

We now build a complete example combining the elements we just saw. Fig. 2 shows a simple analysis which counts accesses to fields and groups them a unique identifier for that field.[14] This analysis could be used to profile hotness of fields, as a basis for cache optimisation.

As is evident from the code, a few minor tweaks have been made. These are all changes which, it turns out, help to make the pieces fit together straightforwardly. We discuss each of these in turn.

Shadow value map Although we could use various Map implementations to hold our analysis's shadow state (line 2 in Fig. 2), we use our own ShadowMap. This is somewhat tailored to analysis use. In particular, ShadowMap additionally allows a factory class to be supplied, enabling get(K key) to create a new map entry in the case of an absent key.

[14] This identifier is defined carefully so as to account for like-named fields occurring within the same class (by inheritance), and for distinct class loaders.

```
1  // shadow values
2  Map<String, AtomicLong> fieldAccesses = new ShadowMap<>(...);
3
4  // (a) first define a stateful mapper which latches the field being accessed
5  class FieldMapper
6    extends ThreadLocal<String>                  // 1. maintain a String per thread
7    implements AfterCompletion<FieldAccess> {    // 2. by hooking field accesses
8      public void afterCompletion(FieldAccess codeRegion)
9      { set(FieldAccessContext.getFullFieldName(codeRegion)); } // grab current field
10 }
11 // (b) then instantiate it
12 FieldMapper currentField = new FieldMapper();
13
14 // update rule: atomic counter increment, from the library
15 Analysis<FieldAccess> updater = new PostIncrement<>(fieldAccesses, currentField);
16
17 // deployment -- infers what instrumentations are needed
18 FRANC.deploy(FRANC.complete(currentField, updater));
```

Fig. 2. Composing a simple field access counter

```
1 class PostIncrement <T extends CodeRegion> implements AfterCompletion<T> {
2   Map<?, AtomicLong> map; MutableReference current;
3
4   public PostIncrement(Map<?, AtomicLong> m, MutableReference c)
5   { map = m; current = c; }
6
7   public void afterCompletion(T codeRegion)
8   { map.get(current.get()).incrementAndGet(); }
9 };
```

Fig. 3. A value updater included in our library—in this case, an atomic incrementer

Mapper The mapper (line 5 in Fig. 2) is essentially unchanged from earlier. Since our analysis groups field accesses by the field's unique identifier, it needs to "latch" this identifier whenever a field is accessed. Since each thread may be accessing (at most) one distinct field at a time, our mapper state is a ThreadLocal string value, recording the unique identifier of the last-accessed field. Recall that this state will be used to arrange that each field access is "mapped onto" a different shadow value (i.e. element in the map) depending on what field is being accessed.

Scheduling interfaces One change is that our mapper implements the AfterCompletion<FieldAccess> marker interface, instead of plain After<FieldAccess>, with the effect that it runs after only those field accesses which complete non-exceptionally.

Context accessors It is worth paying closer attention to how the context information is accessed, in line 9 of Fig. 2, using context accessors. These form a separate class hierarchy than both the scheduling interfaces (such as Before, After) and the code region markers (such as FieldAccess). Here, the FieldMapper calls a static method on FieldAccessContext to extract the field's unique identifier. All access to context is done through such methods, and always passing the current code region, which represents the instrumentation site. This allows us to statically define the relationship between context information and the instrumentation sites at which it is available. As a concrete example, the FieldAccessContext class requires that its codeRegion arguments are in fact FieldAccess instances.

Consequently, even though context is defined in a separate class hierarchy, programmer errors involving inappropriate context selection are caught statically. By contrast, these errors are only caught dynamically under existing systems (such as DiSL, custom instrumentations in Chord, etc.). By extending this approach to custom dynamic context, to be described in §3.3, we also avoid many scenarios in which the programmer would otherwise resort to accessing the operand stack directly—another source of run-time errors.

Updaters For each field access we intercept, we require an update function for the shadow values stored in the map. This is a simple increment. More precisely, it is a *post-increment*, meaning the increment happens *after* the completed execution of the underlying bytecode (here, after each field access). As shown in Fig. 3, our library defines PostIncrement for this purpose, appropriately parameterised. The user needs only to instantiate it. Note that we pass it a reference to our currentField mapper state, and to the map itself. All that the incrementer knows about the mapper is that it defines a method get() which will supply some object (here the field name) usable as a key to index the map.[15] This "thin waist" hourglass design is what allows many distinct updaters to be composed unmodified with many distinct mappers.

Autocompleted instrumentation Previously (§3.1), we saw how to manually instantiate Instrumentation objects which act as sources of field access events. In practice, in this example we do not need to construct an Instrumentation explicitly, since our updater implements Analysis<FieldAccess>; instead, we construct the analysis with FRANC.complete(), which infers (from types of its arguments) that a source of FieldAccess events must be added to the composition. Since there is only one such source defined in the library, it is implicitly added. In some more advanced cases, instrumentations will need to be explicitly constructed.

Ordering So far, a subtle question has gone unanswered. We have two distinct clients of FieldAccess events, and they share state. In what order does their code run? Our example requires that the FieldMapper runs first, so that when the updater calls get(), it sees an up-to-date unique field identifier. Deployment logic (in FRANC.deploy()) embodies a series of rules for deciding this ordering. By default, these arrange that mapper logic runs before updater logic, though the user may also specify an ordering explicitly. We discuss this issue further in §3.5.

3.3 Open, Re-usable Event Definitions

The library defines a large number of specialised CodeRegion marker interfaces such as BasicBlock, FieldAccess, ArrayAccess, Body (meaning method or constructor body), and various others (also including most individual bytecodes). Each represents some feature found in Java bytecode. In all cases, the user can specify

[15] Here, get() is defined by ThreadLocal; see §3.4.

```
1  /* marker superinterface for defining features in bytecode (body, basic block, ...) */
2  public interface CodeRegion {}
3
4  /* superinterface for instrumentation-sensitive code */
5  public interface Analysis<T extends CodeRegion> { /* see Sec. 3.3 */ }
6  /* now, fine-grained interfaces for before/after/...  */
7
8  // runs before bytecode of interest
9  public interface Before          <T extends CodeRegion> extends Analysis<T>
10 { void before(T codeRegion); }
11
12 // run after bytecode of interest completes non-exceptionally
13 public interface AfterCompletion<T extends CodeRegion> extends Analysis<T>
14 { void afterCompletion(T codeRegion); }
15
16 // run after, in exceptional case
17 public interface AfterThrowing <T extends CodeRegion> extends Analysis<T>
18 { void afterThrowing(T codeRegion); }
19
20 // run after, in both exceptional and non-exceptional cases
21 public interface After          <T extends CodeRegion> extends Analysis<T>
22 { void after(T codeRegion); }
```

Fig. 4. Marker "scheduling" interfaces for code invoked at instrumentation sites

code that should run before the feature, after it, and so on, as shown in Fig. 4. A key property of our API design is that the set of events is extensible, and that the information accompanying each event is also (independently) extensible, and that accesses to this information are statically checked. Here we review how this is achieved.

Context information For any event, there will be some context information that is accessible from all occurrences of such an event. For example, the context information for a FieldAccess includes the object in which the field is stored, the field's name and type, and so on. Context information is captured by code defined in classes separate from the instrumentation; we have FieldAccessContext, BasicBlockContext and so on. This separation is significant; it leads a *pull-style* access to context information, rather than the *push-style* of other frameworks. Pull-style access brings modularity advantages; we return to this issue shortly.

New events by composition Many events are defined compositionally. For example, the library defines an Allocation event in terms of the various Java bytecodes that perform allocation: not only the new bytecode, but also newarray, anewarray and multianewarray.[16] Therefore, Allocation is defined as the union of several events. Users may define new events in a similar compositional fashion. (They may also define new events from scratch, at the bytecode level; we describe this in §3.6.)

Custom events and custom context User-defined events may require user-defined context information. For example, just as BasicBlockContext defines methods for getting the size and identifier of the basic block, and AllocationContext defines a getAllocated() method returning the allocated object, other events may wish

[16] In fact, we use ObjectConstructor, the event of executing the method body of java.lang.Object, rather than the new bytecode, because this also catches some allocations within the JVM or native code.

```
1  @Before(marker = BytecodeMarker.class, args = "getfield,putfield,getstatic,putstatic")
2  public static void onFieldAccess(BytecodeStaticContext bcsc, ClassContext cc,
3      FieldAccessStaticContext fasc, DynamicContext dc) {
4    Object owner; boolean isStatic;
5    int access = bcsc.getBytecodeNumber();
6    if(access == Opcodes.GETFIELD) {
7      owner = dc.getStackValue(0, Object.class);
8      isStatic = false;
9    }
10   else if(access == Opcodes.PUTFIELD) {
11     owner = dc.getStackValue(1, Object.class);
12     isStatic = false;
13   }
14   else {
15     owner = cc.asClass(fasc.getDeclaredOwner());
16     isStatic = true;
17   }
18   /* method continues ... */
19 }
```

Fig. 5. Non-reusable DiSL code for extracting the containing object for different cases of field access

to extract other information. These two examples represent different kinds of context: BasicBlockContext is said to be *static context*, because it can be determined at load time, simply by inspecting the bytecode and constant pool of the instrumented code. Meanwhile, AllocationContext requires inspection of the program state (to get the allocated object from the operand stack), so is *dynamic context*.

Push versus pull Custom context is a unique feature of our system: we are aware of no other system which allows users to extract additional information at an instrumentation site without also redefining the instrumentation site itself. For example, in Chord or RoadRunner, we would have to write new code against the underlying ASM or Javassist interfaces to achieve this; in doing so, we would extend the callback signature or Event data type used to pass the information to the client. This is push-style access to context information; the callback or Event definition is widened to encapsulate (hopefully) all information the client might require (else pay the cost of invasive code changes). Pull-style access, by contrast, avoids this need to fully anticipate what context information will be required. Rather, an unmodified instrumentation can effectively be extended simply by defining new context providers. For example, we could extend AllocationContext to provide a getSize() method returning the size of the object allocated, without changing the instrumentation defined by Allocation.

Abstraction over instrumentations A second benefit of our separate context approach is that we can abstract over multiple instrumentation sites, extracting commonality in terms of their context information. Our Allocation example is already an instance of this: many different bytecodes do allocations, and the method for getting a reference to the allocated object varies a little. In previous systems such as DiSL, with support for user-defined events ("markers") but not custom dynamic context, hand-rolled code would be required. Fig. 5 shows

```
// shadow state
Map<Pair<CCTNode, String>, AtomicLong> state = new ShadowMap<>(...);

// subordinate mapper which maintains call chains in a calling context tree
final MethodCCT methodCCT = new MethodCCT(); // from the library

// primary mapper: latch the field being accessed, pairing it with current CC
class FieldCCMapper
  extends ThreadLocal<Pair<CCTNode, String>> // 1. maintain per thread <CC,String> pairs
  implements AfterCompletion<FieldAccess> {  // 2. by hooking field accesses
    public void afterCompletion(FieldAccess codeRegion) {
      set(Pair.valueOf(
        methodCCT.current.get(),             // 3. and sampling the call chain
        FieldAccessContext.getFullFieldName(codeRegion)
      ));
} }
FieldCCMapper currentFieldAndCC = new FieldCCMapper();

// update rule: atomic counter increment, from the library
Analysis<FieldAccess> updater = new PostIncrement<>(state, currentFieldAndCC);

// deployment -- infers any additional instrumentation needed
FRANC.deploy(FRANC.complete(methodCCT, currentFieldAndCC, updater));
```

Fig. 6. Composing a call chain sensitive field access counter

a typical example of this, from an implementation of the Racer race-detection algorithm [4] in the original DiSL. Since this hand-rolled code is written inside the snippet, it is not easily re-used. By contrast, in FRANC equivalent definitions can be in library code (namely FieldAccessContext).[17]

3.4 Separating Mappers from Updaters

As described earlier, a "thin waist" hourglass design is used to separate mappers (like FieldMapper) from updaters (like PostIncrement). We illustrate the key features of this design using a slightly more developed example. We stay with counting field accesses, as in Fig. 2, but wish to make it *context-sensitive*, in the sense of keeping counts per call chain (as well as per field identity). Fig. 6 shows such an analysis.

We note that our currentFieldAndCC (together with its equivalent in Fig. 2, currentField) is effectively a double indirection: it is a reference to an object containing a mutable reference. This is precisely what the standard Java ThreadLocal class implements, although in this case holding one mutable reference per thread. We provide our own ThreadLocal which rebases the standard Java implementation onto a new class hierarchy, in which it is a subclass of MutableReference. Our "thin waist" design is based on this class: the updater requires only that it has access to a MutableReference which records the current key object which

[17] Although we could put the snippet shown in Fig. 5 into a library of such snippets, such a snippet could only be made useful by adding the "push" limitation. To pass on the extracted context information, here owner, would require adding code to make a method call, bringing the same fragility already noted in Chord and RoadRunner.

it should use to index the map (which, again, can be any map).[18] The updater performs the lookup and updates the shadow value it finds.[19]

The map storing the results can equally well have either identity or equality semantics. However, we find identity to be a very useful option when mapping through complex domains, such as call chains in this example (or actually call chains paired with strings). Our library provides multiple tree-based data structures such as our MethodCCT (actually a prefix tree, or trie). These have the property that each node represents a distinct value (call chain), and the identity of the node suffices to signify that value (since each distinct call chain is represented by exactly one node). These prefix-structured key spaces are common in dynamic analyses of structured programs—consider not only call chains, but lock stacks, loop nests and others. Using these representations, key matching becomes a simple reference equality test, making identity-based maps the appropriate choice. This approach to representing values is similar to the interned string pool in the JVM or the autoboxing cache maintained by Integer.valueOf(n), and usually represents a more favourable time–space trade-off than content-based value comparisons. Furthermore, static context information (§3.3) is always stored in the instrumented bytecode's constant pool; strings stored here are always interned by the JVM, making such strings also suitable for use as keys in this way.

3.5 Supporting Common Case Usages: Autocomplete

Each instrumentation instance has a number of *subscribers*: the mappers and updaters that run in response to the instrumentation-generated events. FRANC.complete() infers, given a set of subscribers, any extra instrumentations that must be instantiated, and what subscription relationships to create.

As an example, in Fig. 2, we saw that no FieldAccess instrumentation was explicitly created, but two consumers of this information (both the mapper and the updater) were created. By passing these consumers to FRANC.complete(), the requirement for a FieldAccess instrumentation is inferred automatically.

FRANC.complete() is a variadic function. By deciding the order of the arguments, the programmer controls the ordering in which different subscribers' code will be executed. This is important because mappers and updaters interact by side-effecting updates of mapper state, and may both subscribe to the same event. Typically mappers should run first, so that an updater will make an "up-to-date" selection of shadow value. FRANC.deploy() warns if an updater is scheduled before a mapper. (We propose a more general approach to this issue in §7.)

[18] Here we pay a small price of not statically checking that the MutableReference yields a key object that is type-correct with respect to the map.

[19] We may compare this with the logical view presented in §2, where mappers were depicted as "routing" events from instrumentation to updater. Unlike routers, our mappers do not explicitly forward data; they simply maintain the state used by updaters to select which shadows to update. In effect, mappers implement the "control plane" of a router; the data plane resides in the updater and event subscription logic.

3.6 Supporting Advanced Usages

We saw in §3.3 how custom context information can be extracted from code regions already defined. In some cases, we might wish to define not only new context information, but an entirely new code region. This is also supported. With this ability, we retain the full expressiveness of the underlying DiSL system. However, this functionality is applicable only in the rare cases in which existing code regions, or compositions thereof, do not satisfy the user's requirements. (One example might be superblocks or similar trace-like bytecode sequences, which are not currently implemented in our library.) To do so, the programmer must identify the bytecodes of interest using the underlying instrumentation API.

4 Case Studies

In this section we illustrate the benefits of FRANC by recasting two dynamic program analysis tools from the literature, Racer [4] and Senseo [27]. Racer is a data-race detection tool, while Senseo is a dynamic analysis tool for code comprehension and profiling. Note that **(author?)** [2] describe different implementations of Racer based on AspectJ, while **(author?)** [23] compare implementations of Senseo based on DiSL, AspectJ, and ASM.

4.1 Racer

Racer is a data race detection tool for multi-threaded Java programs. Based on an extension of the Eraser algorithm [28], it reports a data race if two or more threads access the same field without holding any common lock, and if at least one of the threads is writing to the field.

The implementation of Racer addresses three major concerns. First, it maintains a mapping layer from field identifiers to their state. Note that the field identifier is a Pair<Object, String>: the first element identifies the object on which the field has been accessed, while the second contains the field's fully-qualified name. Second, it updates a thread local set of the locks held by each thread. Third, it intercepts field access events and updates some state corresponding to the accessed field. State information includes the threads that accessed the field, the intersection of the sets of locks held upon each access, and the access type (i.e., read or write). Fig. 7 illustrates the FRANC recast of Racer.

Lines 2–15 of Fig. 7 are dedicated to the first concern that we identified in Racer, that is, maintaining the mapping layer. In particular, line 2 defines a Map (i.e., state), while lines 6–15 define its updating function (i.e., mapper).

This mapping layer resembles those illustrated in Fig. 2 and Fig. 6. While the mapping layer in Fig. 2 statically identifies fields with a unique field identifier, the implementation presented in Fig. 7 takes into account also the reference on which the field has been accessed. Note that it is possible to replace the mapping layer in Fig. 2 with the one in Fig. 7 without affecting the rest of the

```
1  // shadow state
2  final Map<Pair<Object, String>, RacerState> state = new ShadowMap<>(...);
3
4  // mapper: latch the reference holding the field being accessed,
5  //         pairing it with the field name
6  class FieldOwnerMapper
7    extends ThreadLocal<Pair<Object, String>>        // 1. maintain a pair per thread
8    implements AfterCompletion<FieldAccess> {        // 2. by hooking field accesses
9      public void afterCompletion(FieldAccess codeRegion) {
10       set(Pair.valueOf(
11         FieldAccessContext.getHolder(codeRegion), // 3. and the field holder
12         FieldAccessContext.getFullFieldName(codeRegion)
13       ));
14 } };
15 final FieldOwnerMapper mapper = new FieldOwnerMapper();
16
17 // lock set analysis, from the library
18 LockSetAnalysis lockSet = new LockSetAnalysis();
19
20 class RacerAnalysis implements AfterCompletion<FieldAccess> {
21   public void afterCompletion(FieldAccess codeRegion) {
22     state.get(mapper.get()).onFieldAccess(
23       Context.getFullMethodName(codeRegion),       // location
24       FieldAccessContext.isFieldRead(codeRegion), // access type
25       lockSet.get()                                // set of held locks
26     );
27 } };
28 RacerAnalysis racer = new RacerAnalysis();
29
30 FRANC.deploy(FRANC.complete(lockSet, mapper, racer));
```

Fig. 7. FRANC recast of the Racer data-race detection tool

code. Moreover, it is possible to combine the mapping layer in Fig. 6 with the one in Fig. 7 by using Tuple<CCTNode, Object, String> as key.

Line 18 of Fig. 7 instantiates a lock set analysis component that is part of the FRANC library. This maintains a thread local set of locks held by each thread, thus addressing the second concern of Racer.

Finally, the third concern of Racer, that of updating the state associated to the accessed field, is addressed in lines 20–28 of Fig. 7. The logic for updating the analysis state is here an essential detail of the analysis, so is of limited reusability. However, we note that good modularity properties remain. For example, replacing the mapping layer would not require any modification to this part of the code.

4.2 Senseo

Senseo is a dynamic analysis tool for code comprehension and profiling. It collects context-sensitive dynamic information for each invoked method. This information consists of (1) the number of method executions, (2) the run-time type of method arguments, (3) the run-time type of return values, and (4) the number of allocated objects. Each of these can be seen as collected by a smaller sub-analysis. The implementation of Senseo addresses two key concerns. Firstly, it must maintain the current call chain for each thread. Secondly, it must perform the necessary analyses upon method entry, method exit, and memory allocation.

Fig. 8 illustrates the FRANC code addressing the first concern, that is, maintaining per-thread call chains. This code is equivalent to part of the code illustrated in Fig. 6. However, here we do not declare a single Map to store the

```
// mapper which maintains call chains in a calling context tree
final MethodCCT methodCCT = new MethodCCT(); // from the library

// (a) first define a stateful mapper which latches the current CC
class CCMapper
  extends MethodLocal<CCTNode> {       // 1. maintains the current CC in a local variable
    public CCTNode initialValue() {    // 2. setting its initial value
      return methodCCT.current.get(); // 3. by sampling the call chain
} };
// (b) then instantiate it
CCMapper currentCC = new CCMapper();
```

Fig. 8. Mapping layer of the FRANC recast of Senseo

```
// shadow state for methodCalls
Map<CCTNode, AtomicLong> callsState = new ShadowMap<>(...);
// update rule: atomic counter increment, from the library
Analysis<Body> methodCalls = new BeforeIncrement<>(callsState, currentCC);
```

(a) Count the number of method executions

```
// shadow state for methodArgs
Map<CCTNode, ArgsProfile> argsState = new ShadowMap<>(...);
// runtime argument type analysis, from the library
Analysis<Body> methodArgs = new ArgumentAnalysis(argsState, currentCC);
```

(b) Profile the runtime type of method arguments

```
// shadow state for methodRets
Map<CCTNode, RetsProfile> methodRetsState = new ShadowMap<>(...);
// runtime return type analysis, from the library
Analysis<Body> methodRets = new ReturnValueAnalysis(methodRetsState, currentCC);
```

(c) Profile the runtime type of return values

```
// shadow state for allocs
Map<CCTNode, AllocsProfile> allocsState = new ShadowMap<>(...);
// allocation analysis, from the library
Analysis<Allocation> allocs = new AllocationAnalysis(allocsState, currentCC);
```

(d) Profile object and array allocations

Fig. 9. Different analyses in the FRANC recast of Senseo

shadow state. Rather, each specific analysis allocates its own Map. In this way, single analyses can be added or removed without affecting the rest of the code.

```
// shadow state for bbs
Map<CCTNode, BasicBlockProfile> bbsState = new ShadowMap<>(...);
// allocation analysis, from the library
Analysis<BasicBlock> bbs = new BasicBlockAnalysis(bbsState, currentCC);
```

Fig. 10. Optional basic block analysis for the FRANC recast of Senseo

Fig. 9 illustrates the different analyses of which Senseo is composed. Each analysis is completely independent from the others, and could be easily disabled or extended. An advantage of this modular design is that adding an additional analysis, for example to count the number of basic blocks of code executed within each call chain, is as trivial as adding the code snippet presented in Fig. 10. Another advantage is that this code does not depend on a specific instrumentation of BasicBlock code regions, but it can be reused for any custom instrumentation

that intercepts code regions that implement the BasicBlock interface. This is particularly important in this case, as developers may want to use custom algorithms to define BasicBlock regions. For example, bytecodes that could throw an exception may (or may not) define the beginning of a new basic block.

5 Performance Evaluation

In this section, we evaluate the performance of the Senseo and Racer case studies reimplemented using the FRANC framework, and compare it to the performance of the same tools implemented using DiSL. We compare the execution times of both tool implementations on the benchmarks from the DaCapo [3] suite (release 9.12), excluding the tradesoap, tradebeans, and tomcat benchmarks due to well known issues[20] unrelated to our framework. We have used the default workload size to evaluate the Senseo tools, and the small workload size to evaluate Racer tools, due to high memory consumption with both the DiSL and the FRANC frameworks. All experiments were run on a multicore platform[21] with all non-essential system services disabled.

We present the results in Fig. 11, both for the startup and the steady-state performance. Each column in the plot corresponds to a single DaCapo benchmark, and we report the overhead factor of a FRANC-based tool over a DiSL-based tool, when applied to that benchmark. In the last column, we report the geometric mean of all overhead factors. The hollow data points correspond to the mean startup overhead, while the filled data points correspond to the mean steady-state overhead. The whiskers represent a 95% confidence interval for the means.

To determine the startup overhead, we executed 3 runs of a single iteration of each benchmark and measured the time from the start of the process till the end of the iteration to capture the instrumentation overhead. We relied on the filesystem cache to mitigate the influence of I/O operations during startup. To determine the steady-state overhead, we made a single run with 15 iterations of each benchmark. Based on visual inspection of the data, we excluded the first 3 iterations to minimize the influence of startup transients and interpreted code.

Since our system considerably raises the level of abstraction for the analysis developers, we expect to pay certain price in terms of performance. The additional complexity of the inlining of event subscribers may cause additional overhead at instrumentation. The composability of elements and the separation between shadow value mapping and updating inevitably leads to more indirections at run time, increasing overhead.

[20] See bug ID 2955469 (hardcoded timeout in tradesoap and tradebeans) and bug ID 2934521 (StackOverflowError in tomcat) in the DaCapo bug tracker at `http://sourceforge.net/tracker/?group_id=172498&atid=861957`.

[21] Intel Core2 Quad Q9650 CPU, 3.0 GHz, 8 GB RAM, 64-bit Ubuntu GNU/Linux 12.04, kernel 3.2.0-20, Oracle Java HotSpot 64-bit Server VM 1.6.0_32, 7 GB heap.

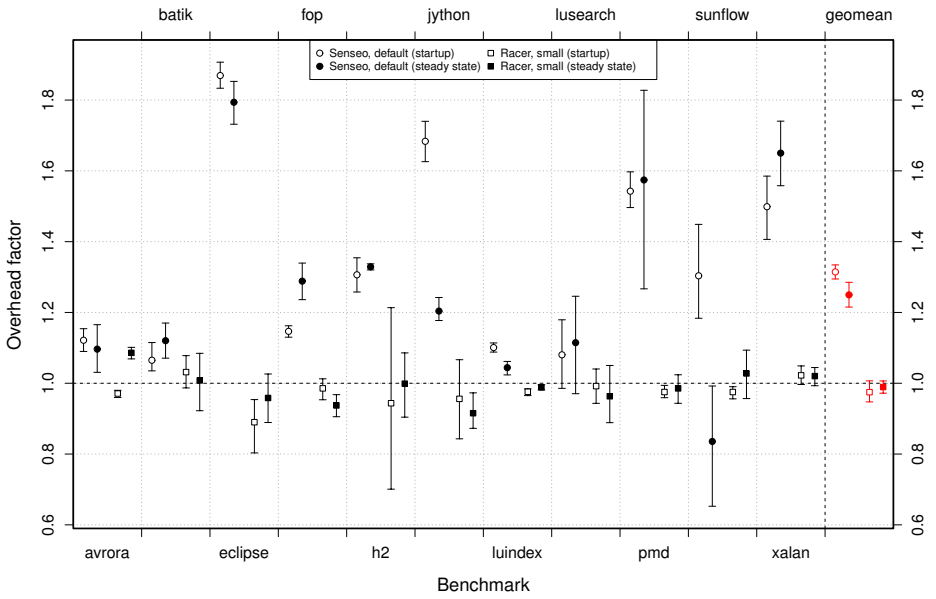


Fig. 11. Mean startup and steady-state overhead, with 95% confidence interval

The results confirm our expectations. While on average, the FRANC-based Racer tool performs on par with the DiSL-based implementation, the FRANC-based Senseo tool shows approximately 31% startup, and 25% steady-state overhead. The average is not entirely representative in the case of Senseo, because there are 3 benchmarks (eclipse, pmd, and xalan) showing considerably higher steady-state overhead, and 4 benchmarks (also including jython) showing considerably higher startup overhead. The absolute worst-case overhead was observed in the eclipse benchmark under the FRANC-based Senseo tool, which yielded an 85% upper bound for the confidence interval.

However, we note that the system is still in a prototype stage, with opportunities for optimization (actually enabled by the higher level of abstraction) still unexploited. A key benefit of our design is that it has been carefully crafted to compile down to code very similar to what an analysis developer would write by hand using a more traditional instrumentation system, such as DiSL (which serves as the foundation for our system). Considering that the DiSL-based Senseo tool was more than twice as fast when compared to an AspectJ-based implementation [23], we consider the overhead found in this evaluation acceptable, and expect the cost of the higher abstraction level to remain in reasonable bounds.

6 Related Work

Basic instrumentations. Several instrumentation systems including Pin [21] and the instrumentation engine of Chord provide callback-based APIs which allow

programmatic construction of analyses in response to a closed set of events. As described in §2, these systems are valuable, but inadequate to modularise complex analyses. DiSL [23] provides slightly more flexibility in defining new instrumentations and extracting additional data ("context information"—§3.3) but retains the same key limitations. Aspect-oriented languages such as AspectJ [18] or AspectC [8] provide analogous support, in the form of join points, but these are again fixed by the system, and lack the low-level (instruction or basic block) join points provided by instrumentation systems.

Event and shadowing abstractions in RoadRunner. RoadRunner is a dynamic analysis framework highly specialised towards data race detection and related analyses. Only per-field, per-lock and per-thread shadowing is provided. Adding new event types would require changes to core interfaces, including the basic `Tool` superclass. By contrast, our system supports an open set of events and a generalised approach to shadowing based on shadow mappers (§3.2), and provides for many common requirements using library code, making it more general but no harder to use. RoadRunner decomposes analyses into an event-processing pipeline, admitting some modularity and re-use. The linear (pipeline) topology of data flow in RoadRunner is also limiting, because decisions made by one pipeline stage affect all subsequent stages; there is no provision for "forking" of pipelines or "fan out" of events, yet many of the more complex use cases require this. (Consider an analysis for comparing the performance of different locking disciplines; we would want to synthesise two distinct streams of locking events, each in a manner similar to RoadRunner's existing `FaultInjection` module, then simulate the performance of each. Unfortunately, RoadRunner's interfaces provide no way to distinguish the two separate streams of lock events.)

Relations in Chord. The Chord framework provides two support mechanisms for the creation of dynamic analyses. The most basic is a set of instrumentation-based callbacks, implemented on top of Javassist [6, 7], as already discussed. The second is a relational storage and query model (based on Datalog) which can be used to store and analyse collected data. These additional abstractions assist the programmer in a manner roughly analogous to our mapper and updater facilities. Each different kind of incoming event can be seen as a distinct "program domain" (in Chord terminology), shadow value maps as Datalog relations, updaters as updates to the contents of a relation, and mappers as join-based queries selecting the elements to update. However, an important difference is that Chord's storage and query infrastructure is shared with its static analysis capabilities, and consequently incorporates far-reaching design choices optimised for scalability of static analysis problems to large input programs. In particular, all analysis happens in a postprocessing stage, rather than being interleaved (or in parallel) with the base program, since relations cannot be queried until each participating domain is fully constructed. For example, a simple profiler counting basic block executions by call chain would defer computing counts until the domain of call chains was fully constructed, i.e. until the end of execution. Although justifiable

within static analysis problems, when used for dynamic analysis they exclude a key mode of application: interactively monitoring running systems.

DTrace. DTrace [5] is a system for dynamic, safe observation of both user and kernel code in production systems. It permits an open set of events (or "probes" in DTrace terminology), defined separately from the framework by distinct *providers* having user-defined semantics and implementations. Implementing a new provider is a very involved process; indeed, one plausible approach could to use a dynamic instrumentation system of the kind we have discussed. DTrace's built-in user-level provider eschews existing dynamic instrumentation systems in favour of source-level macros and link-time interposition, largely to ensure a *zero overhead when disabled* property. (We note that this is a property of a provider, not of DTrace in general.) Analyses in DTrace are written in D, an Awk-like scripting language including some powerful features for storing and aggregating collected data. These include built-in associative mappings, which may be keyed on various kinds of values, including call chains, and a notion of aggregation function which can be used to schedule update operations in a thread-local, contention-minimising fashion. However, D's containers are built-in and its data model is ad-hoc. Our approach is more flexible and more general: our associative containers can be keyed not only on some fixed set of data types, but on any domain that can be constructed by the programmer (such as call graph edges, lock stacks, loop nests, objects, etc.). On the other hand, our system does not share the safety or performance properties that are key constraints on D's design.

Other domain-specific languages. MDL [15] is a domain-specific language for describing performance metrics in terms of instrumentation snippets that compute them. It provides a closed set of join points (including procedure entry/exit and other common cases), *constraints* (turning on or off instrumentation based on the properties of a candidate instrumentation site), and *program resource* definitions which can restrict instrumentation to particular parts of the program. A limited form of composition is supported by combining new constraints with pre-existing, less specialised metrics—but only if that constraint was anticipated by the instrumentation author as being applicable. As well as offering limited compositionality, the system is highly specialised towards performance measurement.

7 Conclusions and Future Work

We have presented FRANC, an API which lifts dynamic analysis construction from the level of instrumentation to an event-based publish–subscribe system with convenient re-usable abstractions for data collection and aggregation.

Although embedding in Java has many benefits, the implicit data flow of an imperative language is a hindrance in many circumstances where we would like to reason explicitly about data flow—for example, to infer the appropriate ordering

between listeners responding to the same event. A fully declarative approach is the logical next avenue to explore. We note that the "signal" abstraction from functional reactive programming fits neatly with our design: it has been argued as an improvement over explicit listener-style publish–subscribe systems [9, 24, 22], and our use of "current" values during shadow mapping (§3.2) is logically the act of sampling a signal. We plan to explore these synergies in the near future.

Acknowledgments. The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the Scientific Exchange Programme NMS–CH (project 11.109), by the European Commission (Seventh Framework Programme projects 287746 and 257414), by the Sino-Swiss Science and Technology Cooperation (SSSTC) Exchange Grant (project EG34–092011) and Institutional Partnership (project IP04–092010), and by the Czech Science Foundation (project 201/09/H057).

References

1. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 85–96. ACM (1997)
2. Ansaloni, D., Binder, W., Villazón, A., Moret, P.: Parallel dynamic analysis on multicores with aspect-oriented programming. In: AOSD 2010: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, pp. 1–12. ACM Press (March 2010)
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: Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA 2006 (2006)
4. Bodden, E., Havelund, K.: Racer: effective race detection using AspectJ. In: Proc. Int. Symp. on Software Testing and Analysis, ISSTA 2008, pp. 155–166. ACM (2008)
5. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proc. USENIX Annual Technical Conference, ATEC 2004, p. 2. USENIX Association (2004)
6. Chiba, S.: Load-time structural reflection in Java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000)
7. Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient Java bytecode translators. In: Pfenning, F., Macko, M. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 364–376. Springer, Heidelberg (2003)
8. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using aspectC to improve the modularity of path-specific customization in operating system code. SIGSOFT Softw. Eng. Notes 26(5), 88–98 (2001)
9. Courtney, A.: Frappé: Functional reactive programming in Java. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 29–44. Springer, Heidelberg (2001)

10. Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. 9th USENIX Conference on Operating Systems Design and Implementation, OSDI 2010, pp. 1–6. USENIX Association (2010)
11. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 121–133. ACM (2009)
12. Flanagan, C., Freund, S.N.: The RoadRunner dynamic analysis framework for concurrent programs. In: Proc. 9th Workshop on Program Analysis for Software Tools and Engineering, PASTE 2010, pp. 1–8. ACM (2010)
13. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. In: Proc. SIGPLAN Symposium on Compiler Construction, SIGPLAN 1982, pp. 120–126. ACM (1982)
14. Hertz, M., Blackburn, S.M., Moss, J.E.B., McKinley, K.S., Stefanović, D.: Error-free garbage collection traces: how to cheat and not get caught. In: Proc. ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, SIGMETRICS 2002, pp. 140–151. ACM (2002)
15. Hollingsworth, J., Niam, O., Miller, B., Xu, Z., Goncalves, M., Zheng, L.: MDL: a language and compiler for dynamic program instrumentation. In: Proc. Conf. Parallel Architectures and Compilation Techniques, pp. 201–212. IEEE (1997)
16. Jovic, M., Hauswirth, M.: Listener latency profiling: Measuring the perceptible performance of interactive java applications. Science of Computer Programming 76(11), 1054–1072 (2011)
17. Jump, M., McKinley, K.S.: Dynamic shape analysis via degree metrics. In: Proc. Int. Symp. on Memory Management, ISMM 2009, pp. 119–128. ACM (2009)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
19. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. Int. Symp. on Code Generation and Optimization, CGO 2004 (2004)
20. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley (1999)
21. Luk, C.-K., Cohn, R., Muth, R., Ptil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proce. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 191–200. ACM (2005)
22. Maier, I., Odersky, M.: Deprecating the Observer Pattern with Scala.react. Technical report, EPFL (2012)
23. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Proc. 11th Int. Conf. on Aspect-Oriented Software Development, pp. 239–250 (2012)
24. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for Ajax applications. In: 24th ACM SIGPLAN Conf. on Object Oriented Programming: Systems Languages and Applications, OOPSLA 2009, pp. 1–20. ACM (2009)
25. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation, pp. 89–100. ACM (2007)

26. Ricci, N.P., Guyer, S.Z., Moss, J.E.B.: Elephant Tracks: generating program traces with object death records. In: Proc. 9th Int. Conf. on Principles and Practice of Programming in Java, PPPJ 2011, pp. 139–142. ACM (2011)
27. Rothlisberger, D., Harry, M., Binder, W., Moret, P., Ansaloni, D., Villazon, A., Nierstrasz, O.: Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. IEEE Transactions on Software Engineering 38(3), 579–591 (2012)
28. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
29. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for c. In: Proc. 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering, ESEC/FSE-13, pp. 263–272. ACM (2005)
30. Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., Zheng, W.: Do I use the wrong definition?: DeFuse: definition-use invariants for detecting concurrency and sequential bugs. In: Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 160–174. ACM (2010)
31. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. 12(1), 157–171 (1986)
32. Weidendorfer, J.: Sequential performance analysis with callgrind and kcachegrind. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 93–113. Springer (2008)
33. Zukowski, J.: The Definitive Guide to Java Swing. 3rd edn. Apress (2005)

AVERROES: Whole-Program Analysis without the Whole Program

Karim Ali and Ondřej Lhoták

David R. Cheriton School of Computer Science
University of Waterloo, Canada
{karim,olhotak}@uwaterloo.ca

Abstract. Call graph construction for object-oriented programs is often difficult and expensive. Most sound and precise algorithms analyze the whole program including all library dependencies. The *separate compilation assumption* makes it possible to generate sound and reasonably precise call graphs without analyzing libraries. We investigate whether the separate compilation assumption can be encoded universally in Java bytecode, such that all existing whole-program analysis frameworks can easily take advantage of it. We present and evaluate AVERROES, a tool that generates a placeholder library that overapproximates the possible behaviour of an original library. The placeholder library can be constructed quickly without analyzing the whole program, and is typically in the order of 80 kB of class files (comparatively, the Java standard library is 25 MB). Any existing whole-program call graph construction framework can use the placeholder library as a replacement for the actual libraries to efficiently construct a sound and precise application call graph. AVERROES improves the analysis time of whole-program call graph construction by a factor of 4.3x to 12x, and reduces memory requirements by a factor of 8.4x to 13x. In addition, AVERROES makes it easier for whole-program frameworks to handle reflection soundly in two ways: it is based on a conservative assumption about all behaviour within the library, including reflection, and it provides analyses and tools to model reflection in the application. The call graphs built with AVERROES and existing whole-program frameworks are as precise and sound as those built with CGC. While CGC is a specific implementation of the separate compilation assumption in the DOOP framework, AVERROES is universal to all Java program analysis frameworks.

1 Introduction

Constructing sound and precise call graphs for object-oriented programs is often difficult and expensive. The key reason is dynamic dispatch: the target of a call depends on the runtime type of the receiver. One approach, Class Hierarchy Analysis (CHA) [8], is to conservatively assume that the receiver could be any object admitted by the statically declared type of the receiver. Because call graphs constructed with this assumption are imprecise, most call graph construction algorithms attempt to track the flow of potential receivers through the

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 378–400, 2013.

program [1,6,12,13,24]. Since the receiver might be created anywhere in the program, these algorithms generally analyze the whole program. However, modern programs have large library dependencies (e.g., the Java standard library). This makes it very expensive to construct a call graph even for a small program. Even if the algorithm itself is cheap, just reading all of the library dependencies of a program takes a long time. Moreover, in many cases, the whole program may not even be available for analysis.

Previously, we defined and evaluated the *separate compilation assumption* [2], which enables a sound and reasonably precise call graph to be constructed for a program without analyzing its library dependencies. In the rest of this paper, we will use the singular "library" to mean all of the libraries that a program depends on. The assumption states that the library is developed and can be compiled without the client program that uses it. This is true of most real programs and their dependencies. The properties that follow from the assumption and from the Java type system effectively limit the imprecision that would otherwise result from conservatively assuming arbitrary behaviour for the unanalyzed library code. We have evaluated the separate compilation assumption in CGC [2], a prototype implementation in Datalog based on the DOOP framework [6]. Our experiments have shown that with the separate compilation assumption, the sound call graphs constructed without analyzing the library can be nearly as precise as those constructed by whole-program analysis. However, implementing the constraints that follow from the assumption in popular analysis frameworks such as DOOP [6], SOOT [24], and WALA [12] is difficult, and would complicate the frameworks significantly and make them more difficult to maintain.

In this paper, we investigate whether the constraints that follow from the separate compilation assumption can be encoded in a form that is universal to all Java program analysis frameworks, the Java bytecode. Our goal is to enable any existing whole-program analysis framework to take advantage of the separate compilation assumption without modifications to the framework. To accomplish this, we present AVERROES, a Java bytecode generator that, for a given program, generates a replacement for the program's library that embodies the constraints that follow from the separate compilation assumption. An existing, unmodified whole-program analysis framework needs only to read the replacement library instead of the original library to automatically gain the benefits of the separate compilation assumption. For example, instead of going through all of the work that was necessary to implement CGC, one can now achieve the same effect automatically by running AVERROES followed by DOOP. Moreover, the same adaptation can be applied automatically not only to DOOP, but to any other whole-program call graph construction framework.

We evaluate the performance improvements that the use of AVERROES enables over the whole-program analysis frameworks SPARK and DOOP. The improvements are very significant because the replacement library is much smaller than the original library: for example, even version 1.4 of the Java standard library contains 25 MB of class files, whereas the AVERROES replacement library contains in the order of only 80 kB of class files. Depending on the size of the

analyzed client program, Averroes improves the running time of Spark and Doop by a factor of 4.7x and 3.7x, respectively, and reduces memory requirements by a factor of 13x and 8.4x, respectively.

Averroes also enables other benefits in addition to performance. One such benefit is generality. For example, many whole-program analysis frameworks are designed to soundly model some specific version of the Java standard library. However, the replacement library constructed by Averroes soundly overapproximates all possible implementations of the library that have the interface used by the client application. Therefore, Averroes makes any existing whole-program analysis framework independent of the Java standard library version. A related benefit is the handling of difficult features such as reflection and native methods. A whole-program analysis must correctly model in detail all such unanalyzable behaviour within the library in order to maintain soundness. On the other hand, Averroes is automatically sound for such behaviour because it already assumes that the library could "do anything". That said, the generated library must still model reflective effects of the library on the client application (e.g., reflective instantiation of classes of the application). However, this issue is also made easier by Averroes. Any tools or analyses that provide information about such reflective effects (e.g., analysis of strings passed to reflection methods or dynamic traces summarizing actual reflective behaviour) can be implemented once and for all in Averroes. Whole-program analysis frameworks can then take advantage of these effects without modification.

The rest of this paper is organized as follows. Section 2 provides background information about call graph construction and the separate compilation assumption. Section 3 describes how Averroes encodes the constraints that follow from the separate compilation assumption in the Java bytecode. Section 4 discusses the performance improvements gained by using the placeholder library generated by Averroes instead of the original library code. Section 5 presents related work, and Section 6 concludes this paper.

2 Background

2.1 Call Graph Construction

A static call graph is an overapproximation of the method calls that may occur in a program at run time. For every method invocation instruction in the program (a *call site*), the call graph contains an *edge* to every *target* method that might be invoked by that instruction. In Java, as well as other object-oriented languages, the targets of method calls are selected using the run-time type of the receiver object. Therefore, call graph construction requires a combination of two static analyses: calculating the sets of possible receiver types (i.e., points-to analysis), and determining the targets of method calls. The two analyses are inter-related: receiver types decide the targets of calls, and the calling relationships between methods determine how objects of specific types flow through the program to the call sites. A precise call graph construction algorithm computes these two

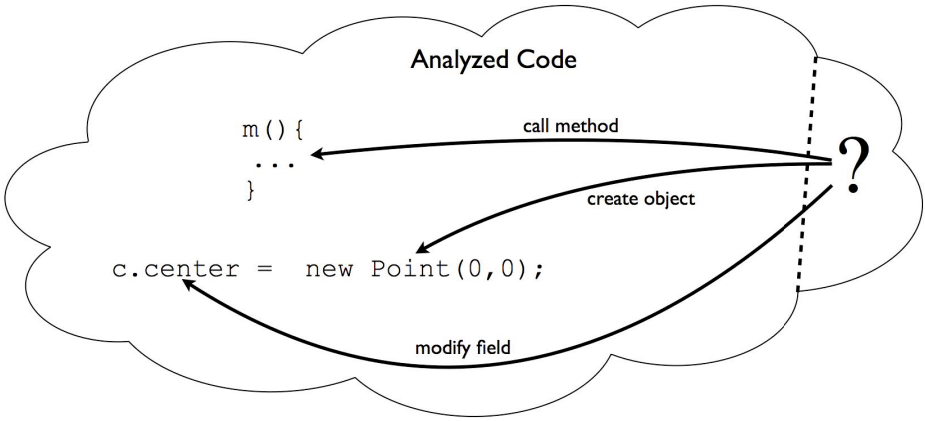


Fig. 1. Conservative assumptions that a sound partial-program analysis must make

inter-dependent analyses concurrently until it reaches a mutual least fixed point. This is sometimes called *on-the-fly* call graph construction.

If the whole program is not available for analysis, a sound call graph construction algorithm must conservatively assume that the unanalyzed code could do "anything". In particular, the unanalyzed code could call any method, assign any value to any field, and create new objects of any type, as summarized in Figure 1. Due to the mutual dependencies between the two analyses that make up any call graph construction algorithm, imprecise results in one analysis can quickly pollute the other. Therefore, without any assumptions about the unanalyzed part of the code, a sound algorithm generates a call graph that is so imprecise that it is useless. However, it is frequently the case that the unanalyzed code is a library that is developed separately and can be compiled without access to the rest of the program. This *separate compilation assumption* [2] enables the construction of a precise and sound call graph for the part of the program that is analyzed (the *application*) without analyzing the library.

2.2 The Separate Compilation Assumption

The key assumption underlying both CGC [2] and AVERROES is that the library can be compiled separately without the client application program. From this assumption, more specific constraints are inferred that bound the possible behaviours of the unanalyzed library code. In CGC, the call graph construction algorithm is extended to conservatively assume that the library can have any behaviour that satisfies the constraints that follow from the separate compilation assumption. AVERROES, on the other hand, constructs a placeholder library that exercises all those behaviours. Any unmodified whole-program call graph analysis can then analyze the application with the placeholder library to achieve a similar result as CGC. The rest of this section briefly summarizes the constraints that follow from the separate compilation assumption and that underlie

Averroes. A thorough discussion of the justification of each of these constraints is found in [2].

Constraint 1 [*class hierarchy*]
A library class cannot extend or implement an application class or interface.

Constraint 2 [*class instantiation*]
An allocation site in a library method can instantiate an object whose run-time type is:
- a library class, or
- an application class whose name is known to the library (i.e., through reflection).

Constraint 3 [*local variables*]
Local variables in the library can point to the following objects:
- objects instantiated by the library,
- objects instantiated by the application and passed to the library due to interprocedural assignments,
- objects stored in fields accessible by the library code, or
- objects whose run-time type is a subtype of `java.lang.Throwable`.

Constraint 4 [*method calls*]
A call site in the library can invoke:
- any method in any library class visible at this call site, or
- a method m in an application class c, but only if:
  1. m is non-static and overrides a (possibly abstract) method of a library class, and
  2. a local variable in the library points to an object of type c or a subclass of c.

Constraint 5 [*field access*]
A statement in the library can access (i.e., read or modify):
- any field in any library class visible at this statement, or
- a field f of an object o of class c created in the application code, if:
  1. f is originally declared in a library class, and
  2. a local variable in the library points to the object o.

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.

Constraint 6 [*array access*]
The library can only access array objects pointed to by its local variables. If the library has access to an array, it can access any of its elements through its index. Similar to field writes, objects written into an array element must be pointed to by a local variable in the library.

Constraint 7 [*static initialization*]
The library causes the loading and static initialization (i.e., execution of the method `<clinit>()`) of classes that it instantiates (according to the class instantiation constraint).

Constraint 8 [*exception handling*]
The library can throw an exception object e if:
- e is instantiated by the library, or
- e is instantiated by the application and passed to the library (as discussed in the local variables constraint).

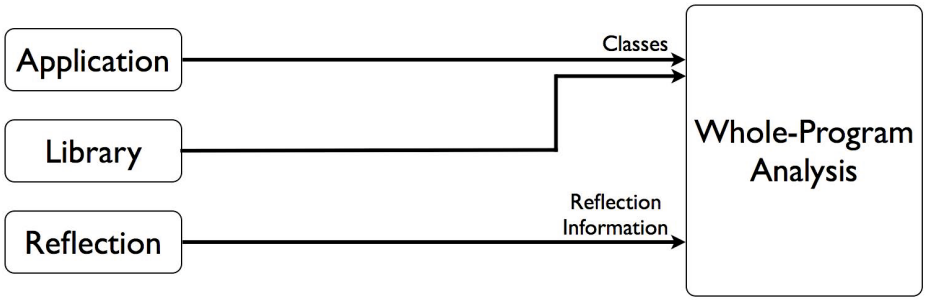


Fig. 2. The usual context of a whole-program analysis

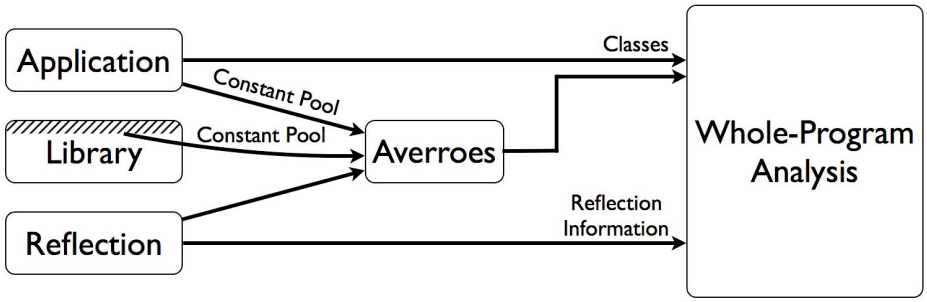


Fig. 3. The context of a whole-program analysis using AVERROES

3 AVERROES Overview

This section presents the context in which AVERROES is used, and defines in detail the contents of the placeholder library that it generates.

The usual context of a whole-program analysis tool is depicted in Figure 2. The tool expects to analyze all of the classes of the program, including any libraries that it uses. The tool does not necessarily distinguish between application and library classes. Optionally, the tool may also make use of additional information about the uses of reflection in the program. This information could be provided by the user or collected during execution of the program being analyzed with a tool such as TAMIFLEX [5].

We have implemented AVERROES, a tool that intends to provide the same input environment to the whole-program analysis, but without analyzing any actual code of the original library classes. We have made AVERROES available at `http://plg.uwaterloo.ca/~karim/projects/averroes/`. Figure 3 depicts the context in which AVERROES is used. Given any Java program, AVERROES generates an alternative placeholder library that models the constraints that follow from the separate compilation assumption. To achieve that, AVERROES uses SOOT [24] to consult the classes of the input program. Unlike a whole-program analysis, AVERROES does not inspect all classes, and does not analyze

any Java bytecode instructions. For each application class, AVERROES examines only the constant pool to find all references to library classes, methods, and fields. Among library classes, AVERROES consults only the classes that are directly referenced by the application and their superclasses and superinterfaces. Within this restricted set of classes, AVERROES examines only the constant pool. AVERROES uses this information in order to build a model of the class hierarchy and the overriding relationships between methods in the program. Since AVERROES examines only a small fraction of classes and only a small fraction of each class file, the execution of AVERROES can be much faster than a whole-program analysis that reads and analyzes the code of the whole program. In addition, if the library code itself calls other dependent libraries, AVERROES can process the library even if those dependencies are not available for analysis, assuming they are not directly referenced from the application code. AVERROES also, optionally, reads in the reflection facts generated by TAMIFLEX [5] for the input program.

The output of AVERROES is a placeholder library. AVERROES again uses SOOT to generate this library. Moreover, AVERROES uses the Java bytecode verification tools offered by BCEL [7] to verify that the generated library satisfies the specifications of valid Java bytecode. The placeholder library contains stubs of all of the library classes, methods, and fields referenced by the application, so that the application could be compiled with the placeholder library instead of the original library. As a consequence, the application classes together with the generated placeholder library make up a self-contained whole program that can be given as input to any whole-program analysis. The placeholder library is designed to be as small as possible, while still being self-contained, such that the whole-program analysis can analyze it much more efficiently than the original library. In addition, the placeholder library overapproximates all possible behaviours of the original library, so that the call graph analysis produces a sound call graph when analyzing the placeholder library instead of the original library. The rest of this section defines in detail the contents of the generated placeholder library.

3.1 Library Classes

The AVERROES placeholder library contains three kinds of classes: referenced library classes, concrete implementation classes, and the AVERROES library class. We define the structure of these classes first, and we define the contents of their methods in Section 3.2.

Referenced Library Classes. The AVERROES placeholder library contains every library class directly referenced by the application and their superclasses and superinterfaces. In addition, it contains a small fixed set of basic classes that are mentioned explicitly in the Java Language Specification [14] and expected by whole-program analyses (e.g., `java.lang.Object` and `java.lang.Throwable`).

Each such referenced class contains placeholders for all of the methods and fields that are referenced by the application. A method m is considered to be referenced by the application if:

- a reference to m appears in the constant pool of an application class,
- m is a constructor or a static initializer in the original library class, or
- a call to some method m' referenced by the application may resolve to m.

A field f of type t is considered to be referenced by the application if a reference to f appears in the constant pool of an application class. If an included library method is native in the original library, its placeholder is made non-native. This is because the generated placeholder library should stand alone and not depend on other code, including native code. Furthermore, the *throws* clause of a generated placeholder library method can only contain exception classes that are referenced by the application. To ensure that every library class has at least one accessible constructor, AVERROES adds a default constructor (i.e., a public constructor that takes no arguments) to every included library class.

Concrete Implementation Classes. The class instantiation constraint of the separate compilation assumption states that the library code can create an object of any library type. This includes types that are not referenced by the application. The object of an unreferenced type could still be accessed by the application through one of its super-types. For the purpose of constructing an application-only call graph, the exact run-time type of the object is not important, since any calls on the object will just resolve to the library summary node. However, the call graph construction algorithm must be aware that an object of such an unknown type could be the receiver of a call.

Figure 4(a) shows a sample Java program that calls the method `java.util.Vector.elements()`. The return type of the method is `java.util.Enumeration`, which is an interface. If the application then calls a method such as `hasMoreElements()` or `nextElement()` on the value returned from `java.util.Vector.elements()`, the call should resolve to the library. Therefore, the call

```
1  class Main {
2    void foo() {
3      Vector v = new Vector();
4      ...
5      Enumeration e = v.elements();
6      while(e.hasMoreElements()) {
7        ...
8      }
9    }
10 }
```

(a)

```
1  class EnumerationConcrete
2        implements Enumeration {
3    boolean hasMoreElements() {
4      return true;
5    }
6
7    Object nextElement() {
8      return (Object) libraryPointsTo ;
9    }
10 }
```

(b)

Fig. 4. An example illustrating the concept of concrete implementation classes in AVERROES: (a) sample application Java code that uses the class `java.util.Enumeration`, (b) the concrete implementation class that AVERROES creates for `java.util.Enumeration` in the placeholder library.

graph construction analysis must be aware that the receiver of the call could be some object that implements the `java.util.Enumeration` interface. However, if the application does not implement the interface itself, and if it does not reference any library class that implements it, then the whole-program analysis would not know about the existence of any concrete class that implements the interface. In this case, AVERROES adds to the placeholder library a concrete class that implements the interface, so that the call graph construction algorithm can resolve the call on this class. Figure 4(b) illustrates the contents of the concrete implementation class that AVERROES generates for `java.util.Enumeration` in the placeholder library.

Specifically, AVERROES creates a concrete implementation class for each interface and abstract class in the library that is referenced by the application, but is not implemented by any concrete class already in the placeholder library. If the original library contains a concrete class implementing the given interface or abstract class, that concrete class would already be in the placeholder library only if the application references that concrete class. Each concrete implementation class contains implementations of all abstract methods in the interface or abstract class that caused the concrete implementation class to be created, including abstract methods inherited from superclasses and superinterfaces.

AVERROES **Library Class.** All of the conservative approximations of the possible behaviours of the library defined by the constraints listed in Section 2.2 are implemented in one class in the placeholder library, `AverroesLibraryClass`. In particular, this class models the following library behaviours: object instantiation, callbacks to application methods, array writes, and exception handling. The `AverroesLibraryClass` has two members:

1. The field `libraryPointsTo` is a public, static field of type `java.lang.Object`. It represents all local variables in the original library code. Every object that could be assigned to a local variable in the original library is assigned to this field. The points-to set of the `libraryPointsTo` field corresponds to the `LibraryPointsTo` set in CGC.
2. The method `doItAll()` is a public, static method. It is the main AVERROES method that models all of the potential side effects that the original library code could have.

3.2 Library Methods

Referenced Library Method Bodies. Each placeholder method in the referenced library classes and in the concrete implementation classes is an entry point from the application into the library, and should conservatively implement the behaviours specified in Section 2.2. Most of these behaviours are implemented just by calling the `doItAll()` method of the `AverroesLibraryClass`. In addition, each placeholder method stores all of its parameters to the `libraryPointsTo` field. The return value of the method is also taken from `libraryPointsTo`.

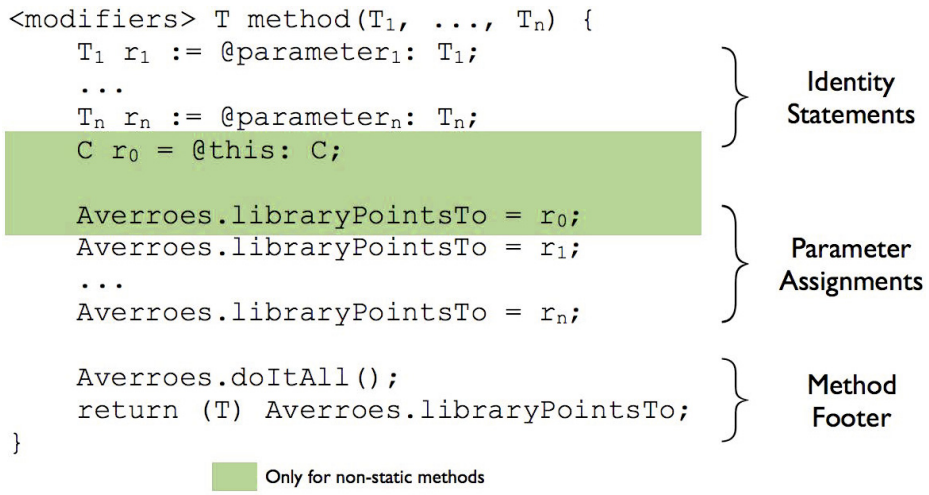


Fig. 5. The Jimple template used by AVERROES to generate bodies for referenced library methods

More specifically, the body of each placeholder method is constructed according to the template shown in Figure 5. The template is shown in the Jimple intermediate language of the SOOT framework [24], which is used by AVERROES to generate the placeholder library. The template has three code regions:

1. Identity statements define the variables that will hold the method parameters. Non-static methods have an additional identity statement for the implicit `this` parameter.
2. Parameter assignment statements assign the parameters to the `libraryPointsTo` field in order to model the interprocedural flow of objects from the application through parameters into the library (the local variables constraint).
3. The method footer contains two statements. The first statement is a call to the `doItAll()` method in the `AverroesLibraryClass` to model the side effects of the library. The second statement is the `return` statement of the method. The method can return any object from the library whose type is compatible with the return type of the method. This is modelled by reading the `libraryPointsTo` field and casting its value to the method return type. This completes the implementation of the local variables constraint. If the return type of the method is primitive, the constant value 1 is returned. Methods with return type `void` will just have an empty `return` statement.

The bodies of constructors of placeholder library classes are generated using the same Jimple template. However, a call to the default constructor of the direct superclass is generated before accessing the `this` parameter in the constructor body. Moreover, AVERROES generates statements that initialize the instance fields of the declaring class. Each instance field is initialized by assigning it the

value of the `libraryPointsTo` field after casting it to the appropriate type (the field access constraint).

The bodies of library static initializers are simpler. Since static initializers have no parameters or return value, no identity statements or parameter assignment statements are generated for them. In addition, they have an empty `return` statement (i.e., one that does not return any value). Moreover, for each static initializer, AVERROES initializes the static fields of its declaring class with the value from the `libraryPointsTo` field through the appropriate cast (the field access constraint).

AVERROES `doItAll()` **Method Body.** The `doItAll()` method implements most of the conservative approximation of the behaviour of the whole library. It is a static method with no parameters, and therefore does not have any identity statements. The body of the `doItAll()` method implements the following behaviours:

1. Class instantiation (Constraints 2 and 7): According to the class instantiation constraint, the library can create an object of any concrete class in the library or any application class that is instantiated by reflection. For each such class c, two statements are generated: a `new` instruction to allocate the object, and a special invocation instruction (corresponding to the `invokespecial` bytecode) to an accessible constructor of the class. Finally, if the class c declares a static initializer, AVERROES generates a call to it.
2. Library callbacks (Constraints 3 and 4): Following the method calls constraint, the `doItAll()` method contains calls to all methods of the library that are overridden by some method of the application, since at run time, any such call could dispatch to the application method. In addition, the `doItAll()` method calls all application methods known to be invoked by reflection. The receiver of all of these calls is taken from the `libraryPointsTo` field, as are all arguments to the method. The values from the `libraryPointsTo` field are cast to the appropriate types as required by the method signature. Additionally, the local variables constraint states that objects may flow from the application to the library due to interprocedural assignments. Therefore, in AVERROES, if the target method of a library call back has a non-primitive return type, its return value is assigned to the field `libraryPointsTo`.
3. Array element writes (Constraint 5): The library could store any object reference that it has into any element of any array to which it has a reference. Two statements are generated to simulate this. The first statement casts the value of the `libraryPointsTo` field to an array of `java.lang.Object`, which is a supertype of all arrays of non-primitive types. The second statement assigns the value of the `libraryPointsTo` field to element 0 of the array.
4. Exception handling (Constraint 8): The library code could throw any exception object to which it has a reference. To model this, AVERROES generates code that casts the value of the `libraryPointsTo` field to the type `java.-`

`lang.Throwable`, and throws the resulting value using the Jimple `throw` statement (which corresponds to the `athrow` bytecode instruction).

In the current implementation of AVERROES, the `doItAll()` method is a single straight-line piece of code with no control flow. If AVERROES were to be used with a flow-sensitive analysis, control flow instructions should be added to all library methods, including the `doItAll()` method. This allows the instructions to be executed in an arbitrary order for an arbitrary number of times. This enables a sound overapproximation for all possible control flow in the original library. Although this would be easy to implement, we have not done it because all of the call graph construction frameworks for Java that we are aware of mainly do flow-insensitive analysis.

Similarly, the `doItAll()` method writes only to element 0 of every array. If a framework attempts to distinguish different array elements, this should be changed to a loop that writes to all array elements. Again, we are not aware of any call graph construction frameworks for Java that distinguish different array elements.

3.3 Modelling Reflection

AVERROES models reflective behaviour in the library in two ways. First, whenever a call site in the application calls a library method, AVERROES assumes that any argument of the call that is a string constant could be the name of an application class that the library instantiates by reflection. For every such string constant that is the name of an application class, AVERROES generates a `new` instruction and a call to the default constructor of the class in the `doItAll()` method.

Second, AVERROES reads information about uses of reflection in the format of TAMIFLEX [5]. TAMIFLEX is a dynamic tool that observes the execution of a program and records the actual uses of reflection that occur. AVERROES then generates the corresponding behaviour in the `doItAll()` method. Alternatively, a programmer who knows how reflection is used in the program could write a sound reflection specification by hand in the TAMIFLEX format. AVERROES generates the following code in the `doItAll()` method to model the reflective behaviour recorded in the TAMIFLEX format:

1. For every class that the TAMIFLEX file specifies as instantiated by `java.lang.Class.newInstance()`, or even just loaded by `java.lang.Class.forName()`, the `doItAll()` method allocates an instance of the class using a `new` instruction, and calls its default constructor.
2. For every unique appearance of `java.lang.reflect.Constructor.newInstance()` in the TAMIFLEX file, the `doItAll()` method allocates an instance of the specified class and calls the specified constructor on it.
3. For every unique appearance of `java.lang.reflect.Array.newInstance()` in the TAMIFLEX file, the `doItAll()` method allocates an array of the specified type.
4. For every unique appearance of `java.lang.reflect.Method.invoke()` in the TAMIFLEX file, the `doItAll()` method contains an explicit invocation of the appropriate method.

Even though these behaviours are triggered by reflection in the original library, the AVERROES placeholder library implements all of them explicitly (non-reflectively) using standard Java bytecode instructions. Therefore, even if the whole-program analysis that follows AVERROES does not itself handle reflection, it will automatically soundly handle the reflective behaviour that AVERROES knows about. This is because AVERROES encodes the behaviour explicitly in the placeholder library using standard bytecode instructions known to every analysis framework.

In addition, the placeholder library still contains the methods that implement reflection in the Java standard library. The `doItAll()` method also contains calls to `java.lang.Class.forName()` and `java.lang.Class.newInstance()` on the value of `libraryPointsTo` cast to `java.lang.String`. Therefore, if the whole-program framework knows about the special semantics of these reflection methods, or if it knows about some reflective behaviour that is unknown to AVERROES, the whole-program framework can still model the additional reflective behaviour in the same way as if it were processing the original library instead of the AVERROES placeholder library.

3.4 Code Verification

The placeholder library that is generated by AVERROES is intended to be standard, verifiable Java bytecode that can be processed by any Java bytecode analysis tool. To guarantee this, AVERROES verifies the generated placeholder library classes using the BCEL [7] verifier. BCEL closely follows the class file verification process defined in the Java Virtual Machine Specification [14, Section 4.9]. BCEL ensures the validity of the internal structure of the generated Java bytecode, the structure of each individual class, and the relationships between classes (e.g., the subclass hierarchy).

4 Evaluation

We evaluate how well AVERROES achieves the goal of enabling whole-program analysis tools to construct sound and precise call graphs without analyzing the whole library. First, we quantify the improvements in performance when AVERROES is used with both SPARK and DOOP. Second, we compare the resulting call graphs with dynamically observed call graphs to provide partial evidence that the static call graphs are sound. Third, we compare the call graphs constructed using AVERROES to those constructed using CGC to support the claim that AVERROES enables existing whole-program analysis implementations to perform the kind of analysis that is prototyped in CGC.

We conducted our experiments on two benchmark suites: the DaCapo benchmark programs version 2006-10-MR2 [4], and the SPEC JVM98 benchmark programs [20]. All of these programs are analyzed with the Java standard library from JDK 1.4 (jre1.4.2_11). We ran all of the experiments on a machine with four dual-core AMD Opteron 2.6 GHz CPUs (running in 64-bit mode) and 16 GB of RAM.

We created an artifact for the experiments that we conducted to evaluate AVERROES. The artifact includes a tutorial with detailed instructions on how to use AVERROES to generate the placeholder libraries for each program in our benchmark suites. It then shows how to reproduce all of the statistics we discuss in this section. We have made the artifact available at `http://plg.uwaterloo.ca/~karim/projects/averroes/tutorial.php`. The artifact has been successfully evaluated by the ECOOP Artifact Evaluation Committee and found to meet expectations.

4.1 Performance

To evaluate how much work a whole-program analysis saves by using AVERROES, we first compare the size of the generated placeholder library with the size of the original Java standard library. We then measure the reductions in execution time and memory requirements of both SPARK and DOOP when using AVERROES.

AVERROES **Placeholder Library Size.** Over all of the benchmark programs that we have experimented with, the average size of the input library is 25 MB (min: 25 MB, max: 30 MB, geometric mean: 25 MB), while the average size of the generated AVERROES library is only 80 kB (min: 20 kB, max: 370 kB, geometric mean: 80 kB). Additionally, the average number of methods in the original input library is 36,000 (min: 19,462, max: 48,610, geometric mean: 35,615), while the average number of methods in the generated AVERROES library is only 600 (min: 137, max: 3,327, geometric mean: 570). That means that the number of methods in the placeholder library is smaller by a factor of 62x (min: 13x, max: 286x, geometric mean: 62x). As we will see, this reduction in the library size significantly reduces the time and memory required to do whole-program analysis.

> **Finding 1:** The placeholder library generated by AVERROES is very small compared to the original Java standard library.

Execution Time. We have compared the execution times of both SPARK and DOOP when analyzing each benchmark with the AVERROES placeholder library and the original Java standard library. We break the total time required to construct a call graph into three components. First, the AVERROES *library generation time* is the time required for AVERROES to inspect the application for references to the library and to generate the placeholder library. Second, the *overhead time* is the time required for SPARK or DOOP to prepare for call graph construction analysis. In the case of SPARK, this preparation includes reading the whole program from disk and constructing internal data structures. In the case of DOOP, this preparation additionally includes generating the constraints required for the analysis and encoding them in Datalog relations. Third, the *analysis time* is the time required for SPARK or DOOP to solve the constraints and generate a call graph.

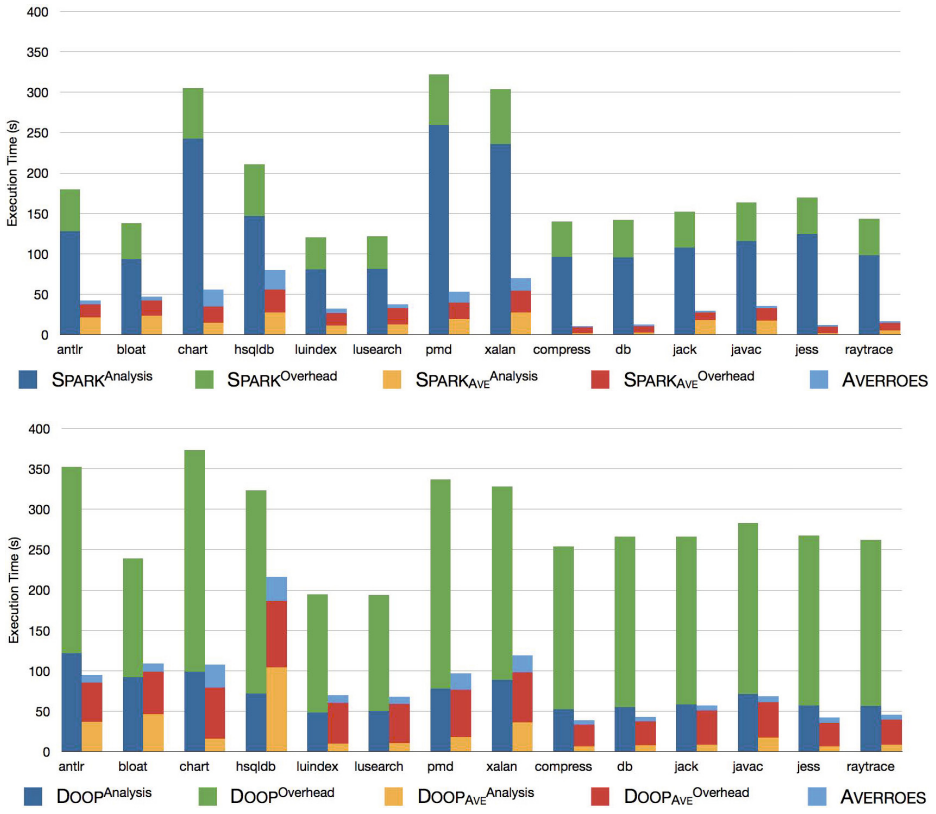


Fig. 6. The execution time of whole-program tools (SPARK and DOOP) compared to AVERROES-based tools ($\text{SPARK}_{\text{AVE}}$ and DOOP_{AVE})

Figure 6 compares the times required for call graph construction by SPARK and DOOP with the original Java library (denoted SPARK and DOOP) and with the AVERROES placeholder library (denoted $\text{SPARK}_{\text{AVE}}$ and DOOP_{AVE}). AVERROES reduces the analysis time of SPARK by a factor of 12x (min: 4x, max: 62x, geometric mean: 12x) and of DOOP by a factor of 4.3x (min: 0.7x, max: 9.3x, geometric mean: 4.3x). In general, whole-program analysis is expensive not only because of the analysis itself, but also due to the overhead of reading a large whole program from disk and pre-processing it. Replacing the large Java library with the much smaller AVERROES placeholder library reduces the time that SPARK executes (including overhead and analysis time) by a factor of 6.8x (min: 3.3x, max: 17x, geometric mean: 6.8x), and the time that DOOP executes by a factor of 4.3x (min: 1.7x, max: 7.7x, geometric mean: 4.3x). When the AVERROES library generation time is added to the time taken by SPARK or DOOP to finish, the total overall time to execute $\text{SPARK}_{\text{AVE}}$ is faster than SPARK by a factor of 4.7x (min: 2.5x, max: 10.3x, geometric mean: 4.7x), and the total overall time

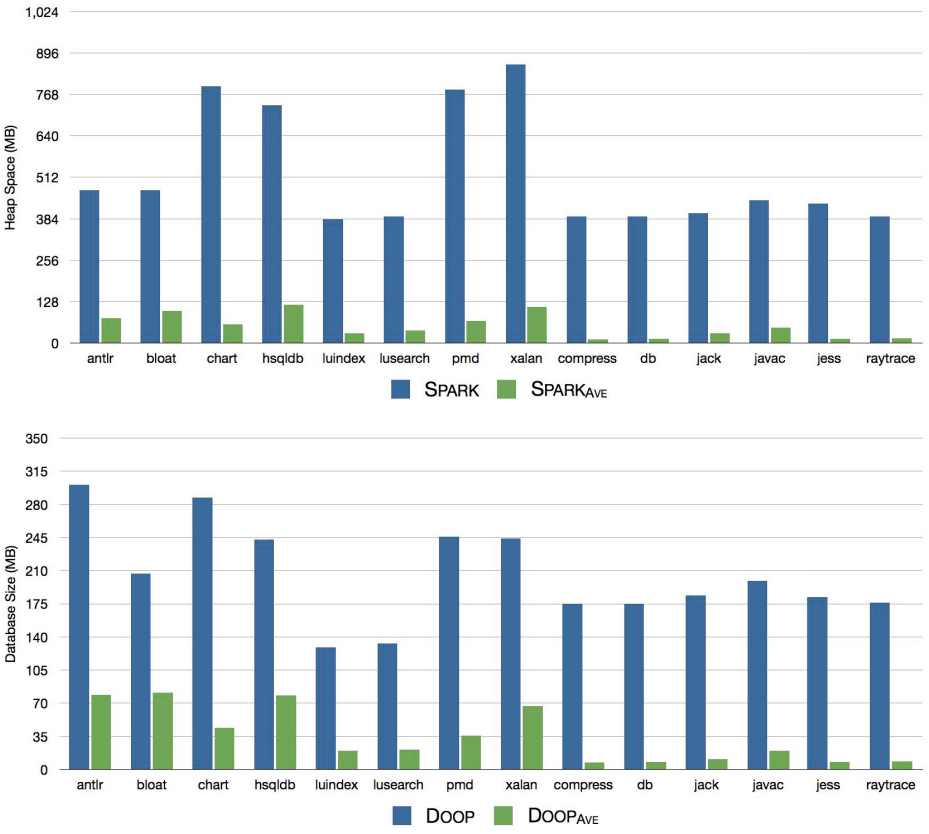


Fig. 7. The memory requirements of whole-program tools (SPARK and DOOP) compared to AVERROES-based tools ($\text{SPARK}_{\text{AVE}}$ and DOOP_{AVE})

to execute DOOP_{AVE} is faster than DOOP by a factor of 3.7x (min: 1.5x, max: 6.5x, geometric mean: 3.7x).

Finding 2: AVERROES enables whole-program tools to construct application call graphs faster.

Memory Requirements. SPARK and DOOP store their intermediate results in different ways. SPARK does all the calculations in memory, while DOOP stores intermediate facts in a LogicBlox [15] database on disk. Therefore, we use different methods of calculating the memory requirements of each tool. We compare the maximum amount of heap space used during call graph construction by $\text{SPARK}_{\text{AVE}}$ and SPARK. On the other hand, we compare the on-disk size of the database of relations computed by DOOP_{AVE} and DOOP.

Figure 7 compares the memory usage of SPARK$_{\text{AVE}}$ against SPARK, and DOOP$_{\text{AVE}}$ against DOOP. Overall, SPARK$_{\text{AVE}}$ requires 13x less heap space than SPARK (min: 4.8x, max: 35x, geometric mean: 13x), and DOOP$_{\text{AVE}}$ uses 8.4x less disk space than DOOP (min: 2.6x, max: 24, geometric mean: 8.4x).

Finding 3: Using AVERROES reduces the memory requirements of whole-program analysis tools.

4.2 Call Graph Soundness

Static call graph construction is made difficult in Java by dynamic features such as reflection, and features that are difficult to analyze such as native methods. AVERROES makes it much easier to construct a sound call graph in the presence of these features in two ways. First, whereas whole-program analysis frameworks try to model all behaviour of the whole program precisely, AVERROES uses the conservative assumption that the library could have any behaviour consistent with the separate compilation assumption. Therefore, a whole-program analysis must model every detail of dynamic behaviour or risk becoming unsound. On the other hand, an analysis using AVERROES remains sound without having to precisely reason about dynamic behaviour within the library. Second, AVERROES contains analyses that model how the library affects the application using reflection. These analyses make use of information about strings passed into the library, as well as information about reflection generated by TAMIFLEX [5]. A whole-program analysis that uses AVERROES can automatically benefit from the results of these analyses without having to implement the analyses themselves.

We have evaluated the soundness of static call graphs by comparing them against dynamic call graphs collected by *J [10]. Since a dynamic call graph results from only a single execution, it may miss edges that could execute in other executions. Therefore, such a comparison does not guarantee that the static call graph is sound for all executions. Nevertheless, the comparison can detect soundness violations, and the lack of detected violations provides at least partial assurance of soundness. The results of this comparison are shown in Table 1. The DYNAMIC line shows the number of call edges in the application portion of the dynamic call graph. The remaining lines show how many of these edges are missing in the static call graphs generated by SPARK and DOOP with and without using AVERROES. When using AVERROES, only two edges are missing from all of the call graphs. In `lusearch`, a `NullPointerException` is thrown and the dynamic call graph records a call edge from the virtual machine to the constructor of this exception class. This behaviour is not modeled by either SPARK or DOOP. In `xalan`, a call edge to `java.lang.ref.Finalizer.register()` from the application is missing from the call graph generated by SPARK$_{\text{AVE}}$ since SPARK does not handle calls to this library method. On the other hand, the call graphs generated by SPARK and DOOP without using AVERROES are missing a significant number of dynamically observed edges in benchmarks that make heavy use of reflection. This is despite the immense effort that has been expended to make these analysis frameworks handle reflection soundly.

Table 1. Comparing the soundness of AVERROES-based tools to the whole-program tools with respect to the dynamic call graphs

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
DYNAMIC	3,449	4,257	657	1,627	726	539	2,087	2,953	43	54	596	2,538	13	330
DYNAMIC-SPARK	0	0	0	61	4	185	3	96	0	0	0	0	0	0
DYNAMIC-SPARK$_{\text{AVE}}$	0	0	0	0	0	1	0	1	0	0	0	0	0	0
DYNAMIC-DOOP	0	0	0	331	303	241	225	349	0	0	0	0	0	0
DYNAMIC-DOOP$_{\text{AVE}}$	0	0	0	0	0	1	0	0	0	0	0	0	0	0

Finding 4: AVERROES reduces the difficulty of constructing sound static call graphs in the presence of reflection.

4.3 Comparison with CGC

We previously evaluated the soundness and precision of call graphs constructed by CGC [2]. Our empirical evaluation showed that the static call graphs from CGC are sound when compared against the corresponding dynamic call graphs. Our results also showed that CGC generates precise static call graphs when compared against those generated by SPARK and DOOP, for most programs in our benchmark suite. However, there are spurious edges in the call graphs generated by CGC. Further investigation showed that most of these spurious edges are due to spurious library callback edges. A library callback edge is an edge from a call site in the library back to a method in the application. Those spurious library callback edges eventually cause a small number of spurious call edges within the application and from the application to the library.

The design goal of AVERROES is to enable existing whole-program analysis frameworks to build call graphs without analyzing the library in the manner of CGC. We validate this claim by comparing the call graphs constructed by CGC with those constructed by DOOP with AVERROES, since CGC is more similar to DOOP than to SPARK. Since the separate compilation assumption overapproximates the targets of call sites in the library, we focus our comparison on the library callback edges in the call graph.

Table 2 shows the number of library callback edges in the call graph generated by CGC but missing from the call graph generated by DOOP$_{\text{AVE}}$ (denoted by CGC-DOOP$_{\text{AVE}}$), and vice versa (denoted by DOOP$_{\text{AVE}}$-CGC). The table also shows the percentage of those missing edges with respect to the total number of edges in the call graph from DOOP$_{\text{AVE}}$. The biggest difference is 2% of the edges, in the `chart` benchmark. All of the edges missing in DOOP$_{\text{AVE}}$ and present in CGC (i.e., DOOP$_{\text{AVE}}$-CGC) are due to more precise handling of reflective constructor calls in AVERROES than in CGC. When the library reflectively creates an object of an application class C, CGC considers the library to potentially call all public constructors of C. On the other hand, AVERROES generates a call edge only to the specific constructor that is actually invoked according to TAMIFLEX. Further investigation shows that some edges are missing in CGC and present in DOOP$_{\text{AVE}}$ (i.e., CGC-DOOP$_{\text{AVE}}$) due to calls of `java.lang.reflect.Constructor.newIn-`

Table 2. Comparing DOOP$_{\text{AVE}}$ with CGC with respect to library callback edges

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC-DOOP$_{\text{AVE}}$	2	0	0	16	0	0	5	44	0	0	0	0	0	0
CGC-DOOP$_{\text{AVE}}$ (%)	0.03%	0%	0%	0.15%	0%	0%	0.1%	0.35%	0%	0%	0%	0%	0%	0%
DOOP$_{\text{AVE}}$-CGC	7	13	54	8	10	17	9	10	0	1	0	4	0	0
DOOP$_{\text{AVE}}$-CGC (%)	0.1%	0.08%	2.02%	0.08%	0.83%	0.68%	0.17%	0.08%	0%	1.33%	0%	0.05%	0%	0%

`stance()`. Whereas CGC ignores these calls (and handles only calls to `java.lang.Class.newInstance()`), AVERROES models calls to both `newInstance()` methods.

Finding 5: AVERROES matches and slightly exceeds CGC in both precision and soundness.

5 Related Work

5.1 Call Graph Construction

A distinguishing feature of different whole-program call graph construction algorithms is how they approximate the targets of dynamically dispatched method calls. This affects how they approximate the run-time types of the receivers of those calls.

Early work on call graph construction used simple approximations of run-time types. Dean et al. [8] formulated *class hierarchy analysis* (CHA), which uses the assumption that the run-time type of a receiver could be any subtype of its statically declared type at the call site. Thus, CHA uses only static type information, and does not maintain any points-to sets of the possible run-time types of objects. Bacon and Sweeney [3] defined *rapid type analysis* (RTA), which refines the results of CHA by restricting the possible run-time types only to classes that are instantiated in the reachable part of the program.

Diwan et al. [9] presented more precise call graph construction algorithms for Modula-3 that remain simple and fast. Rather than maintaining a single set of possible run-time types in the whole program, as in RTA, they compute separate sets of run-time types for individual local variables.

Sundaresan et al. [21] introduced *variable type analysis* (VTA). VTA generates subset constraints to model the possible assignments between variables within the program. It then propagates points-to sets of the specific run-time types of each variable along these constraints. Unlike the analyses of Diwan et al. [9], VTA computes these points-to sets for heap-allocated objects in addition to local variables.

Tip and Palsberg [22] studied a range of call graph construction algorithms in which the scope of the points-to sets was varied between a single set for the whole program (like RTA) and a separate set for each variable (like VTA). Their implementation was later used by Tip et al. [23] to implement Jax, a practical application extractor for Java.

Several static analysis frameworks for Java now include call graph construction implementations with a range of algorithms that can be configured for the desired trade-off between precision and analysis cost. The SOOT [24], WALA [12], and DOOP [6] frameworks all construct call graphs as prerequisites to the other interprocedural analyses that they perform. All three frameworks use whole-program analysis to build the call graph. However, SOOT and WALA can be configured to ignore parts of the input program and generate an unsound partial call graph only for the part of the program that is analyzed. AVERROES enables these and other whole-program frameworks to construct sound partial call graphs.

5.2 Partial-Program Analysis

The excessive cost of analyzing a whole program has motivated various efforts to construct call graphs while analyzing only part of the program.

The analysis of Tip and Palsberg [22] analyzes partial programs by defining a special points-to set, S_E. This set summarizes the objects passed into the unanalyzed external code (i.e., the library). Similar to CGC [2] and AVERROES, the analysis assumes that the external library code can call back an application method if: the application method overrides a library method; and the set S_E contains an object on which dynamic dispatch would resolve to that application method.

The main challenge of analyzing the application part of a program while ignoring the library is determining objects that may escape from the predefined application scope to the library, and vice versa. This directly affects the points-to sets of the local variables in the application and the library (or the summarized library points-to set in the case of partial-program analyses). Grothoff et al. [11] presented Kacheck/J, a tool that is capable of identifying accidental leaks of heap object abstractions by inferring the *confinement* property [25,26,27] for Java classes. Kacheck/J considers a Java class to be *confined* when objects of its type do not escape its defining package. A partial-program analysis then needs only to analyze the defining package of the input Java classes to infer their confinement property.

Rountev and Ryder [18] proposed a novel whole-program call graph construction analysis for C. Although the analysis requires the whole program, it analyzes each module of the program separately. A C program can take the address of a function, and later invoke it by dereferencing the resulting function pointer. Therefore, the function that is invoked depends on the target of the function pointer. The analysis proceeds in two steps. First, conservative assumptions are made about all possible applications that could use a given library. The analysis then builds up a set of constraints that model the precise behaviour of the library. Second, these constraints are used to model the library in an analysis of a specific application. Rountev et al. [17] then adapted the approach to Java. Like AVERROES, their implementation encodes the constraints collected from the library in executable placeholder code. However, unlike AVERROES, this placeholder code is a precise and detailed summary of the exact effects of the library,

and its construction requires the entire library to be analyzed. Moreover, some of the constraints require changes to the application code in addition to the placeholder library. In contrast, the purpose of Averroes is to generate a minimal library stub that enables a sound analysis of the original application code.

Rountev et al. [16] applied a similar approach to summarize the precise effects of Java libraries for the purpose of the interprocedural finite distributive subset (IFDS) and interprocedural distributive environment (IDE) algorithms. Although these algorithms already inherently construct summaries of callees to use in analyzing callers, they had to be extended in order to deal with the library calling back into application code. This is done by splitting methods into the part before and after an unknown call. Summaries are then generated for each part rather than the whole method. When the target of the unknown call later becomes available, the partial summaries are composed. Rountev et al. [19] evaluated the approach on two instances of IDE, a points-to analysis and a system dependence graph construction analysis.

Averroes builds on our work on Cgc [2], which defined the separate compilation assumption and derived from it specific constraints that conservatively model all possible behaviours of the library. These constraints were implemented in Cgc as an extension of an existing whole-program call graph construction tool. Experimental results showed that the resulting call graphs are sound and quite precise compared to those constructed by a whole-program analysis that analyzes the whole library precisely. Whereas Cgc required significant implementation effort to extend the whole-program framework, Averroes enables the same approach to be implemented directly by any existing whole-program call graph construction framework without requiring extensions.

6 Conclusions

We have shown that the separate compilation assumption can be encoded in the form of standard Java bytecode. This enables any existing whole-program call graph construction framework to easily make use of it. Our Averroes generator, given an input program, automatically generates a conservative replacement for the program's library that embodies the constraints that follow from the separate compilation assumption (i.e., a placeholder library).

Constructing the placeholder library is fast and does not require analyzing the whole program. Moreover, the resulting placeholder library is very small, especially when compared to the size of the Java standard library. We have empirically shown that using Averroes with an existing whole-program analysis framework reduces the cost of call graph construction by a factor of 4.3x to 12x in analysis time and 8.4x to 13x in memory requirements. Averroes also makes it easier for a whole-program framework to soundly handle reflection. That is because Averroes makes a conservative approximation of all library behaviour, including reflection. Additionally, Averroes provides support for modelling uses of reflection in the application. Finally, we have shown that the call graphs generated with Averroes and an existing, unmodified, whole-program framework are

as precise and sound as those obtained by explicitly implementing the separate compilation assumption in some specific framework.

We plan to extend this work to generate placeholder libraries for various widely-used Java frameworks (e.g., Android, J2EE, Eclipse Plug-in) using AVERROES. We hope that this will lead to an easier means of analyzing client applications developed in these frameworks without the need to analyze the framework itself. Like a library, a framework typically satisfies the separate compilation assumption because it is developed without knowledge of the client applications that will be developed within it. One major difference is that in a framework, the main entry point to the program resides in the framework rather than in the client application. The application is then reflectively started by the framework code. We expect that with only minor changes, AVERROES will be applicable to these and other Java frameworks.

References

1. Agrawal, G., Li, J., Su, Q.: Evaluating a demand driven technique for call graph construction. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 29–45. Springer, Heidelberg (2002)
2. Ali, K., Lhoták, O.: Application-only call graph construction. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 688–712. Springer, Heidelberg (2012)
3. 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)
4. 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)
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. Dahm, M., van Zyl, J., Haase, E.: The bytecode engineering library (BCEL) (November 2003), `http://commons.apache.org/bcel/`
8. 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)
9. Diwan, A., Moss, J.E.B., McKinley, K.S.: Simple and effective analysis of statically-typed object-oriented programs. In: 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 1996, New York, NY, USA, pp. 292–305 (1996)

10. 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)
11. 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)
12. IBM: T.J. Watson Libraries for Analysis WALA (November 2012), `http://wala.sourceforge.net/`
13. 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)
14. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
15. LogicBlox Home Page (April 2013), `http://logicblox.com/`
16. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 2–16. Springer, Heidelberg (2006)
17. 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)
18. Rountev, A., Ryder, B.G.: Points-to and side-effect analyses for programs built with precompiled libraries. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 20–36. Springer, Heidelberg (2001)
19. 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)
20. Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (May 2012), `http://www.spec.org/jvm98/`
21. 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)
22. 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)
23. 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)
24. 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)
25. Vitek, J., Bokowski, B.: Confined types. In: 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 82–96 (1999)
26. Vitek, J., Bokowski, B.: Confined types in Java. Softw., Pract. Exper. 31(6), 507–532 (2001)
27. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. J. Funct. Program. 16(1), 83–128 (2006)

QUIC Graphs: Relational Invariant Generation for Containers

Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan

University of Colorado Boulder
{arlen.cox,evan.chang,sriram.sankaranarayanan}@colorado.edu

Abstract. Programs written in modern languages perform intricate manipulations of containers such as arrays, lists, dictionaries, and sets. We present an abstract interpretation-based framework for automatically inferring *relations* between the set of values stored in these containers. Relations include inclusion relations over unions and intersections, as well as quantified relationships with scalar variables. We develop an abstract domain constructor that builds a container domain out of a *Quantified Union-Intersection Constraint* (QUIC) graph parameterized by an arbitrary *base domain*. We instantiate our domain with a polyhedral base domain and evaluate it on programs extracted from the Python test suite. Over traditional, non-relational domains, we find significant precision improvements with minimal performance cost.

1 Introduction

Container manipulating programs are ubiquitous. Essentially all high-level programming languages provide a standard library with container types, such as arrays, lists, dictionaries, and sets. In this paper, we investigate static analysis techniques for inferring assertions about the possible set of values that can be stored in containers at run time. Our analysis abstracts containers by the set of elements contained in them to infer facts about (a) the possible set of values in a container; and (b) how these values relate to values stored in other containers. In general, one may envision two main types of static analyses: (1) *content-centric* analyses that infer assertions for the possible sets of values in each container, *in isolation*; or (2) analyses that infer relations between the values stored in various containers, *as-a-whole*.

To illustrate this difference, consider the Python code function extendClass in the inset (the name of this function will become clearer below). This function takes a set X and returns a set Y where Y is the subset of elements from X such that each element is greater than or equal to some variable c. An important post-condition of extendClass is $Y \subseteq \{\nu \in X | \nu \geq c\}$, but neither the content-centric nor the as-a-whole analyses can produce this post-condition. The content-centric analysis, which represents each set X and Y as individual variables in a domain for reasoning about values, would produce $Y \subseteq \{\nu | \nu \geq c\}$ where ν ranges over the universe of values. Because

```
def extendClass(X):
  Y = set([x for x in X
       if x >= c])
  return Y
```

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 401–425, 2013.

all values of X are not related to all values of Y in some way, a content-centric analysis cannot represent any relationship between X and Y. As-a-whole analyses, which reason only about the relationships between sets, can produce $Y \subseteq X$, but fail to infer anything about the individual elements of Y. By combining these two classes of analyses, our analysis finds the desired invariant: $Y \subseteq \{\nu \in X | \nu \geq c\}$.

```
def extendClass(D):
  E = {k:v for k,v in a.iteritems()
       if k >= "%"}
  return E
```

The extendClass function is abstracted from a function in Processing.js[1]. A simplified version of the original function is shown in the inset (in Python); the original set version models the key set of this dictionary version. This function copies a dictionary containing a number of values to another dictionary. It only copies those elements that start with letters higher than % in the ASCII table, specifically excluding keys starting with $. These dictionaries are used as objects, and in the context of this framework, $ is interpreted as private and thus should not be copied. Functions like this one are pervasive in programs written in dynamic languages because most run-time structures are implemented using dictionaries (or objects, maps, or tables) and those run-time structures are directly accessible by the developer and can be modified. As a result, previously simple operations such as inheriting a class become complex dictionary manipulations involving copy operations. To statically analyze programs written in dynamic languages, we require powerful new static analysis techniques that can reason about these kinds of functions.

Our analysis tracks (subset) inclusion relations between expressions involving set abstractions of containers through a special graph structure called a QUIC graph. A QUIC graph is a succinct encoding of *set expressions* and *inclusion relations* between them. The expressions represented by a QUIC graph are (1) basic, *atomic sets* that abstract the set of values stored in a container and singletons created by scalar expressions; (2) restricted unions and intersections of the atomic sets; and (3) comprehensions of set expressions through first-order predicates. The predicates are captured by an arbitrary base domain, which can reason about program variables and formal bound variables that represent the scalar-valued contents of a basic set. The QUIC graph is thus a compact structure for storing a conjunction of subset constraints between set expressions. In this paper, we define QUIC graphs and build abstract domain operations over these graphs. The QUIC graph domain is designed to yield a tight integration between the base domain and the QUIC graph domain so that the resulting analysis can transfer facts from one domain to another, quite seamlessly.

The content-centric analysis of containers is rather well understood (e.g., [7, 11, 13]). Such analyses focus on strategies for partitioning or splitting *summary variables* that smash the contents of the container into an essentially weak-updated scalar variable. These techniques are orthogonal and complementary to our work here. With summary variables, one might capture independent comprehensions, such as $X \subseteq \{\nu | p(\nu)\} \wedge Y \subseteq \{\nu | p'(\nu)\}$ for some predicates p and p'. If the predicates p and p' are the same or related, then these facts may indirectly

[1] `http://processingjs.org/`

imply a relation between X and Y but essentially only through their contents. On the flip side, the pure container-as-a-whole approach would track relations directly between X and Y without characterizing their contents. Some existing containers-as-a-whole approaches incorporate some fixed content reasoning (e.g., [23]). In this paper, we present a tight integration of these two approaches with domains for reasoning about scalar variables and their relations to the set elements. As a result, the QUIC graph domain promises to be a lot more powerful than a simple conjunction of both individual domains.

We have implemented the QUIC graph domain for a simple imperative programming language with integers and sets (of integers). The language captures basic arithmetic over integers and operations over sets such as union, intersections, difference, insertion/deletion of elements, and iteration over sets. We implemented analyzers using the QUIC graph domain, as well as two domains representing the content-centric and container-as-a-whole approaches. The evaluation was carried out by translating a variety of set manipulating programs from the Python test suite. The results are quite promising: the QUIC graph domain is more precise than the other domains, proving more properties than a simple combination of a content-centric approach and a container-as-a-whole approach.

Contributions: In this paper, we make the following contributions:

- We identify the need for simultaneous reasoning about containers as-a-whole *and* their contents to enable modular, precise reasoning of container-manipulating programs (Sect. 2).
- We describe QUIC graphs to represent universally-*Q*uantifed *U*nion and *I*ntersection set *C*onstraints in a canonical manner using a hypergraph data structure. We build an abstract domain (functor) based on QUIC graphs. A novel aspect of our domain is the use of predicate edge labels to capture set comprehensions (Sect. 3).
- We present a framework for *inference* using QUIC graphs. We show how to utilize the structure of QUIC graphs to compute all logical implications of a given QUIC graph. We present the inference procedure for strengthening base domain invariants within a QUIC graph. Finally, we show how laziness significantly improves the cost of inferring consequences of QUIC graphs, and describe an efficient implementation (Sect. 4).
- We define an abstract domain using QUIC graphs with inference and show how all domain operations and reductions are easily implemented using lazy inference (Sect. 5).
- We evaluate the effectiveness of our abstract domain on a set of benchmarks from the Python test suite. We find that for a reasonable performance overhead, our abstract domain is significantly more precise than either a content-centric or a container-as-a-whole approach and unlike the content-centric and container-as-a-whole approaches can automatically prove most properties specified in the Python test suite for set operations (Sect. 6).

$$
\begin{aligned}
\text{program} &::= \text{decl}^* \ \text{stmt}^* \\
\text{decl} &::= \textbf{int}\ \text{scalarVar} \mid \textbf{set}\ \text{setVar} \\
\text{stmt} &::= \text{scalarVar}\ \textbf{:=}\ \text{scalarExpr} \\
&\mid \text{setVar}\ \textbf{:=}\ \text{setExpr} \\
&\mid \textbf{loop}\ \text{stmt}^* \\
&\mid \textbf{branch}\ \text{stmt}^*\ \textbf{orelse}\ \text{stmt}^* \\
&\mid \textbf{havoc}\ \text{setVar} \\
&\mid \textbf{assume}\ \text{conditional} \mid \textbf{assert}\ \text{conditional} \\
\text{setExpr} &::= \emptyset \mid \{\text{scalarExpr}\} \mid \text{setVar} \mid \text{setExpr} \cup \text{setExpr} \\
&\mid \text{setExpr} \cap \text{setExpr} \mid \text{setExpr} \setminus \text{setExpr} \\
\text{scalarExpr} &::= \text{scalarVar} \mid \text{scalarConst} \mid \text{scalarUnary}(\text{scalarExpr}) \\
&\mid \text{scalarBinary}(\text{scalarExpr}, \text{scalarExpr}) \mid \textbf{choose}(\text{setExpr}) \\
\text{conditional} &::= \text{scalarConditionals} \mid \text{setExpr} \subseteq \text{setExpr} \mid \text{scalarVar}\ \textbf{in}\ \text{setVar} \\
\text{setVar} &::= \mathrm{X}, \mathrm{Y}, \mathrm{Z} \\
\text{scalarVar} &::= \mathrm{x}, \mathrm{y}, \mathrm{z} \\
\text{scalarConst} &::= \mathrm{c}
\end{aligned}
$$

Fig. 1. An imperative, set-manipulating programming language. A sequence of a symbol α is written as α^*.

2 Overview

In this section, we walk through inferring the desired post-condition for the extendClass example from Sect. 1 to highlight the main challenges in obtaining precise combined content-as-a-whole invariants that motivate our design of the QUIC graph domain. At a high-level, deriving the desired post-condition for the extendClass function requires the careful application of transitive closure of inclusion constraints, an effective reduction [6] strategy with base domain elements, and a non-trivial join operator.

2.1 Set Language

We assume an imperative programming language with scalar values and set values whose elements are scalars, shown in Figure 1. We assume scalar operations (e.g., addition, subtraction, multiplication, and division) are given as unary or binary operators (scalarUnary or scalarBinary, respectively). For convenience, we fix a single scalar type (integers) in our language. Unless otherwise mentioned, sets are assumed to range over this type (integers). However, our framework is quite general. Because we only assume the base domain is a sound abstract domain, we can handle a variety of types including integers, floats, and strings by using base domains designed to reason over scalar variables of those types. We do not address sets of sets or complex structures such as lists in this paper. However, our framework can be extended to handle these types by instantiating with more complex base domains such as another domain for sets.

For the purposes of analysis, we take an input program and lower the program to introduce additional instrumentation variables. The lowering converts all loops (e.g., **for-in**) into a single non-deterministic **loop** construct and all

```
1  def extendClass(X) {
2    Y := ∅;
3    for (x in X) {
4      if (x > c) {
5        Y := Y ∪ {x};
6      }
7    }
8    return Y;
9  }
```

```
1  def extendClass(X) {
2    Xo := ∅; Xi := X; Y := ∅;
3    loop {
4      assume Xi ≠ ∅;
5      x := choose(Xi); Xi := Xi \ {x};
6      branch {
7        assume x > c; Y′ := Y ∪ {x};
8        Y := Y′;
9      }
10     orelse {
11     }
12     Xo := Xo ∪ {x};
13   }
14   assume Xi = ∅;
15   return Y;
16 }
```

Fig. 2. Left: the extendClass example that filters positive elements from a set X into a set Y. Right: its lowered version.

conditional statements into a non-deterministic **branch** construct. The **havoc** statement is an arbitrary value assignment for modeling unknown effects, and the **assume** statement is used to encode the conditions in each branch. One key instrumentation transforms each **for-in** loop over a set X to introduce two sets Xo, Xi that are assumed to partition X (i.e., $\mathrm{X} = \mathrm{Xi} \uplus \mathrm{Xo}$). The set Xo represents all variables that have been iterated over thus far. Likewise, Xi represents the elements of X that remain to be iterated over. The iteration order is assumed to be non-deterministic. The loop exits when $\mathrm{Xi} = \emptyset$ or alternatively $\mathrm{Xo} = \mathrm{X}$. We assume that iterations over a set X do not modify X in the body of the loop (as is the standard semantics for container iteration).

Example 1. Fig. 2 (left) shows a translation of the Python extendClass example from Sect. 1 to an imperative, set-manipulating program. This program filters elements from an input set X greater than or equal to c into a set Y. The set Y is a variable introduced in the translation to name the set being constructed by the comprehension. The lowered version of this program is also shown alongside (right).

2.2 Motivating Example

In Fig. 3, we annotate the lowered version of the extendClass from Fig. 2. At program point 1, set Xi is initialized to X, while Xo and Y are initialized to the empty set $\emptyset$. The extendClass loop begins at point 2. An arbitrary element x is chosen out of set X at point 4 with the **choose** statement and removed from set Xi. The element x is added to set Y in the first case (point 6) of the non-deterministic **branch**, while set Y is left unchanged in the other (point 9). The

```
     def extendClass(X) {
1      Xo := ∅; Xi := X; Y := ∅;
2      loop {
3            [ X = Xi ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c} ]
4        assume Xi ≠ ∅; x := choose(Xi); Xi := Xi \ {x};
5        branch {
6          assume x > c; Y' := Y ∪ {x};
7.a              [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c}
                   ∧ Y' = Y ∪ {x}                          ∧ x > c ]
7.b              [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c}
                   ∧ Y' = Y ∪ {x} ∧ {x} = {ν ∈ {x} | ν > c} ∧ x > c ]
           Y := Y';
8            [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo ∪ {x} | ν > c} ∧ x > c ]
         }
         orelse {
9            [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c} ]
         }
10       Xo := Xo ∪ {x};
11     }
12     assume Xi = ∅;
13         [ Y ⊆ {ν ∈ X | ν > c} ]
       return Y;
     }
```

Fig. 3. Inferring QUIC graph invariants on the extendClass example

final assignment in the loop (at point 10) simply moves the element x into set Xo to continue the iteration.

The boxed formulas in Fig. 3 are program invariants that we infer (under the pre-condition true), that is, the fixed-point result of an abstract interpretation. Our goal is to be able to derive the post-condition $Y \subseteq \{\nu \in X | \nu > c\}$, that is, output set Y is a subset of the positive elements of the input set X, at program point 13. Here and in the rest of this paper, we use ν as the bound variable for all comprehensions. In this figure, we selectively show the key constraints needed to derive this post-condition. We first observe that although inclusion constraints plus comprehension expressions are sufficient to state the desired post-condition, the inferred loop invariant at point 3 requires a more expressive set expression language (i.e., union expressions). It is straightforward to see that this loop invariant $X = Xi \cup Xo$ along with the loop exit condition $Xi = \emptyset$ implies the desired post-condition and that the initial state where $Xi = X \wedge Xo = Y = \emptyset$ implies the loop invariant.

Let us consider the fixed point iteration of the loop (i.e., showing that loop invariant is inductive and thus consecutes) and focus on the transition to invari-

ant 7.a—the difference with respect to the loop invariant is shown shaded. This transition begins with the addition of element x to set Y. The **assume** is reflected in the invariant with a base domain constraint $x > c$ shown to the right in the box. It is necessary to transfer the relationship between Y and Xo to Y' and Xo to generate the desired function post-condition. Knowing when to transfer these relationships by transitivity is critical to both performance and precision. The QUIC graph representation allows us to limit the guesswork of when to apply the various transitivity rules to derive additional facts.

In invariant 7.b, we show a reduction step that transfers information from the base domain to the QUIC graph domain. In particular, we have that $x > c$, so it is also the case that $\forall \nu \in \{x\} \,.\, \nu > c$ (i.e., applying a $\forall$-introduction rule). In terms of QUIC graphs, we have that any constraint of the form $\{x\} \subseteq \left\{\nu \in \bar{T} | B[\nu]\right\}$ can be strengthened to $\{x\} \subseteq \left\{\nu \in \bar{T} | B[\nu] \wedge \nu > c\right\}$ where B is a predicate described by the base domain and $\bar{T}$ is any basic set expression, including $\{x\}$. For abstract interpretation, the conjunction $\wedge$ becomes a meet operator $\sqcap$ on base domain elements. Thus, we have that $\{x\} \subseteq \{\nu \in \{x\} | \nu > c\}$ as shown in invariant 7.b. This "seed" constraint is sufficient to derive other ones, such as $\{x\} \subseteq \{\nu \in Y' | \nu > c\}$, by transitivity on demand. The QUIC graph structure with singleton known-scalar sets enables an eager transfer of information from the base domain coupled with lazy propagation of this information (see Sect. 4).

This reduction step is used for deriving the invariant at point 8. At point 8, we show the invariant derived from 7.b by projecting out Y (and then renaming Y' to Y). From invariant 7.b, we can intuitively see that $Y' \subseteq \{\nu \in Xo | \nu > c\} \cup \{\nu \in \{x\} | \nu > c\}$ by applying transitivity (and that union with any set is monotonic), so we have that $Y' \subseteq \{\nu \in Xo \cup \{x\} | \nu > c\}$, which gets to our desired result after projecting the old Y and renaming Y' to Y. It is not difficult to check this step; rather, the main challenge in an automated analysis is guessing that these are the appropriate steps to obtain the desired invariant. For example, both $Y' \subseteq \{\nu \in X \cup \{x\} | \nu > c\}$ and $Y' \subseteq Xo \cup \{x\}$ are sound over-approximations of the projection that are syntactically close, but now we have lost too much precision to get our desired post-condition. From the QUIC graph perspective, this derivation is a propagation of facts across nodes and edges that can be done on demand by the *lazy closure* (see Sect. 4).

The invariant at point 9 in the unchanged case entails the invariant at which we just arrived at point 8 (except for the base domain constraint), so the result of the join at program point 10 is the invariant at point 8 without the base domain constraint $x > c$, and after the assignment, we get exactly the loop invariant at point 3.

In summary, it is difficult to derive enough constraints via transitivity and strong enough ones via reduction from the base domain. On the flip side, transitive closure, even with restricted union and intersection constraints, is exponential (see Sect. 4). The QUIC graph representation eases this tension by representing inclusion constraints over unions, intersections, and comprehensions in a canonical manner that facilitates on-demand propagation of information.

3 QUIC Graphs

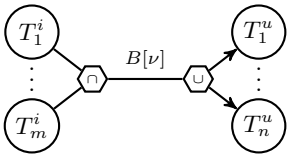
A *Quantified Union/Intersection Constraint graph* is a graph data structure that represents inclusions between set expressions. Throughout the rest of the paper we use the notation X, Y, Z with subscripts to represent set variables and x, y, z with subscripts to represent base domain variables. The special variable ν will be used as a formal bound variable for set comprehensions, as will be explained in this section. The symbol T represents *atomic set expressions* – one of three possible elements: the empty set $\emptyset$, a singleton set containing a base domain variable $\{x\}$ or a set variable X. The symbols $\bar{T}^i, \bar{T}^u$ represent a number of Ts in an intersection or a union respectively.

Definition 1 (QUIC edge). *Let* $T_1^i, \ldots, T_m^i = \bar{T}^i$ *and* $T_1^u, \ldots, T_n^u = \bar{T}^u$ *be symbols representing finite sets and* B *be a base domain abstract state involving a bound variable* ν*, acting as a predicate where* $\top$ *is* true *and* $\bot$ *is* false*. A* QUIC edge *is a constraint*

$$\bigcap_{i=1}^{m} T_i^i \subseteq \left\{ \nu \in \bigcup_{j=1}^{m} T_j^u \;\middle|\; B[\nu] \right\} \text{ represented using the notation } \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_B$$

which is an edge in an edge labeled hypergraph.

We use dots above the set operators simply to make clear that they are part of the syntax of a QUIC constraint or a QUIC edge. Graphically a QUIC edge is represented as a hyperedge:



For convenience, if there is only one T in the union (respectively intersection), we elide the union (respectively intersection) hex from the figure. Additionally, if the label $B[\nu]$ is top in the base domain, we elide the label from the edge.

Definition 2 (QUIC graph). *A QUIC graph* $G \in \widetilde{G}$ *is an edge labeled hypergraph constructed of QUIC edges. It represents a conjunction of constraints where each constraint corresponds to one QUIC edge in the graph. A QUIC graph has the following syntax:*

$$\begin{aligned} G ::= \;& G_1 \;\wedge\; G_2 \\ \mid \;& \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_B \end{aligned}$$

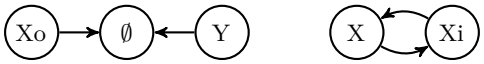
A QUIC graph is a canonical representation of the set of conjoined constraints. It is designed to be compact and to allow efficient inference operations (see Sect. 4).

We provide a series of examples to demonstrate the QUIC graph representation.

Example 2 (Basic QUIC graphs). Consider that would be produced after line 1 from the example in Fig. 3:

$$\mathrm{Xo} \subseteq \emptyset \ \wedge \ \mathrm{Xi} \subseteq \mathrm{X} \ \wedge \ \mathrm{X} \subseteq \mathrm{Xi} \ \wedge \ \mathrm{Y} \subseteq \emptyset$$

This is represented as a QUIC graph:



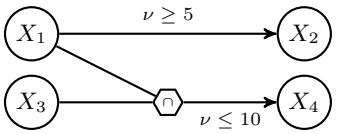
Unlike the constraint formula, the symbols X, Xi, and $\emptyset$ only occur once in the graph. This makes the relationships more clear and eliminates possible redundancy.

Conjoining multiple constraints produces a QUIC graph with multiple edges and including unions or intersections requires a hypergraph to show the relationships:

Example 3. We wish to encode the formula:

$$\dot{\bigcap} X_1 \dot{\subseteq} \dot{\bigcup} X_2 \Big|_{\nu \geq 5} \wedge \dot{\bigcap} X_1, X_3 \dot{\subseteq} \dot{\bigcup} X_4 \Big|_{\nu \leq 10}.$$

We draw this using the following hypergraph:



To be practical, a representation for set constraints cannot stand alone. There must be a way to represent relationships between sets and base domain variables as well. To do this we construct a combined domain where elements are pairs $(G, B) \in \widetilde{S} = \widetilde{G} \times \widetilde{B}$ where G is a QUIC graph domain instance and B is a base domain instance. Note that the base domain has two roles: (a) it labels edges in the QUIC graph and (b) it captures invariants on base domain variables.

To specify the concretization for both QUIC graphs and QUIC graphs combined with an external base domain, a concretization (where γ is overloaded for all concretizations) for the base domain is required:

$$\gamma : \widetilde{B} \to \mathcal{P}((\textsc{BaseVar} \to \textsc{BaseVal}) \times \mathcal{P}(\textsc{BaseVal}))$$

The symbol BaseVar is all base domain variables, BaseVal is all base domain values. This is a non-standard concretization because given some abstraction, it returns a set of functions that map base domain variables to base domain values and for each function, there is a corresponding set that contains the base domain values to which the bound variable ν can be assigned. This is used to define concretization for QUIC graphs

Definition 3 (Concretization). *The concretization γ of a QUIC graph G has the following type, given that* SetVar *is all set domain variables:*

$$\gamma : \widetilde{G} \to \mathcal{P}((\textsc{SetVar} \to \mathcal{P}(\textsc{BaseVal})) \times (\textsc{BaseVar} \to \textsc{BaseVal}))$$

Where the result is a set of pairs of functions (η, η_B), where η maps set variables to sets of base domain values and η_B maps base domain variables to base domain values. These two functions mappings are valid with respect to constraints both on the sets and on the base domain.

The concretization function is then defined as such:

$$\gamma(G_1 \wedge G_2) \stackrel{def}{=} \{(\eta, \eta_B) | (\eta, \eta_B) \in \gamma(G_1) \text{ and } (\eta, \eta_B) \in \gamma(G_2)\}$$

$$\gamma\left(\dot{\bigcap} [T_1^i, \cdots, T_n^i] \dot{\subseteq} \dot{\bigcup} [T_1^u, \cdots, T_m^u]\Big|_B\right) \stackrel{def}{=}$$

$$\left\{ (\eta, \eta_B) \;\middle|\; \begin{array}{l} (\eta_B, \bar{b}) \in \gamma(B) \text{ and} \\ \text{for all } \nu.\ \left(\nu \in \eta(T_1^i) \text{ and } \cdots \text{and } \nu \in \eta(T_n^i)\right) \\ \qquad \text{implies } \left(\nu \in \bar{b} \text{ and } (\nu \in \eta(T_1^u) \text{ or } \cdots \text{or } \nu \in \eta(T_m^u))\right) \end{array} \right\}$$

The concretization for a combined domain S is the same set of pairs (η, η_B), so the type and concretization follow:

$$\gamma : \widetilde{G} \times \widetilde{B} \to \mathcal{P}((\textsc{SetVar} \to \mathcal{P}(\textsc{BaseVal})) \times (\textsc{BaseVar} \to \textsc{BaseVal}))$$

$$\gamma((G, B)) \stackrel{def}{=} \left\{(\eta, \eta_B) | (\eta, \eta_B) \in \gamma(G) \text{ and } (\eta_B, \bar{b}) \in \gamma(B)\right\}$$

Expressivity: We now discuss the expressivity limitations of QUIC graphs. As such, QUIC graphs allow unions, intersections and comprehensions of sets but in a restricted manner. We motivate some of our design choices here.

The first expressivity restriction arises from the manner in which comprehension is introduced in our language. For instance, we are able to express inclusions of the form $X \subseteq \{\nu \in Y \mid B[\nu]\}$ through a QUIC edge. However, QUIC graphs as presented here cannot express the reverse inclusions of the form $\{\nu \in X | B[\nu]\} \subseteq Y$. There are two main reasons for this restriction: (a) Representing reverse inclusions requires a new type of edge relation along with fresh reduction rules for this edge. Additionally, there are many interactions between this new type of relation and existing relations that need to be captured. (b) Reverse inclusions require an abstract domain that implements the underapproximate semantics whereas the inclusions used in QUIC graphs use the standard overapproximate abstract semantics. This ensures that existing abstract domains can be integrated with QUIC graphs without introducing new domain operations. A full theory of QUIC graphs that captures both types of relations will be tackled in the future.

The other expressivity limitation arises from the introduction of union and intersection operations. Note that the relation $X \cup Y \subseteq Z$ can be equivalently expressed simply as $X \subseteq Z \ \wedge \ Y \subseteq Z$. Likewise the intersection $Z \subseteq X \cap Y \ \Leftrightarrow \ Z \subseteq X \ \wedge \ Z \subseteq Y$. This motivates the direction of the union and intersection

Set Operation		Operation using Union/Intersection
$X \subseteq Y \uplus Z$	$\Leftrightarrow$	$X \subseteq Y \cup Z \ \wedge \ Y \cap Z \subseteq \emptyset$
$Y \uplus Z \subseteq X$	$\Leftrightarrow$	$Y \subseteq X \ \wedge \ Z \subseteq X \ \wedge \ Y \cap Z \subseteq \emptyset$
$X \subseteq Y \setminus Z$	$\Leftrightarrow$	$X \subseteq Y \ \wedge \ X \cap Z \subseteq \emptyset$
$Y \setminus Z \subseteq X$	$\Leftrightarrow$	$Y \subseteq X \cup Z$

Fig. 4. Encoding set difference and disjoint union in QUIC graphs

hyperedges in QUIC graphs. We do not directly represent relations between nested unions and intersections unless special existentially quantified variables are permitted in the graph.

Example 4. For instance, the relation $(X_1 \cup X_2) \cap X_3 \subseteq X_4$ cannot be expressed unless a special existentially quantified set variable X_5 is introduced with the constraints

$$\dot{\bigcap} X_5 \mathrel{\dot{\subseteq}} \dot{\bigcup} X_1, X_2 \Big|_\top \wedge \dot{\bigcap} X_1 \mathrel{\dot{\subseteq}} \dot{\bigcup} X_5 \Big|_\top$$
$$\wedge \dot{\bigcap} X_2 \mathrel{\dot{\subseteq}} \dot{\bigcup} X_5 \Big|_\top \wedge \dot{\bigcap} X_5, X_3 \mathrel{\dot{\subseteq}} \dot{\bigcup} X_4 \Big|_\top$$

Finally, relations involving disjoint unions and set difference can also be represented directly using QUIC graph as shown in Figure 4.

Self Loops: Self-loops on QUIC graphs are quite useful to encode assertions that are true of the contents of X in relation to the scalar program variables $x_1, \ldots, x_n$.

Example 5. Let X be a set and x be a program variable. We wish to express that every element in X is between x and $x+10$. We do so in the QUIC graph domain using the self-loop from X to itself labeled by the assertion $\nu \geq x \ \wedge \ \nu \leq x + 10$. In effect, the loop represents the containment relation written

$$X \subseteq \{\nu \in X \mid \nu \geq x \ \wedge \ \nu \leq x + 10\} \quad \text{or} \quad \forall \nu \in X.\, \nu \geq x \ \wedge \ \nu \leq x + 10\,.$$

QUIC graphs naturally represent relationships between set variables, singleton sets and the empty set. However, QUIC graphs do not necessarily represent all possible relationships. In the next section, we show how to derive other relationships from those already in a QUIC graph.

4 Closure

The *closure* of a QUIC graph adds all of the implied logical facts to both the QUIC graph and the base domain. Most of the domain operations of a QUIC graph are defined in terms of the closure by the application of inference rules to add edges to a QUIC graph and strengthen the existing edge labels.

Inference rules are shown in full in Fig. 5. We use three judgment forms. One states when given a combined domain of a QUIC graph and a base domain, $S = (G, B)$, a particular containment relationship is derivable. If the relationship is derivable, the inference judgment provides a predicate B_e that holds on that relationship. The judgment takes the form

$$(G, B) \vdash \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e} ,$$

where $\bar{T}^i$ is the set of intersected vertices and $\bar{T}^u$ is the set of unioned vertices. This judgment relies on an auxiliary judgment $B \vdash x = y$ where x and y are base domain variables. This judgment states when an equality between variables is derivable from a base domain element (and is supplied by the base domain). We also define a judgment $(G, B) \vdash x = y$ that states when an equality can be derived from set constraints.

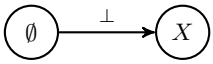
4.1 Inference Rules

We now explain the inference rules for QUIC graphs in detail. A brief explanation of the rules follow.

The (Emp) inference rule generates QUIC graph edges from the empty set to any node, labeled with the bottom base domain element $\bot$ (i.e., with the $\emptyset$ concretization or is equivalent to the predicate **false**).

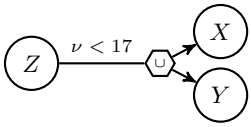
Example 6. Consider the QUIC graph G

By applying (Emp), we get the QUIC graph G':



The (Self-Loop) and (Self-Prop) inference rules generate and strengthen the labels present on self loops in QUIC graphs. The strengthening takes information from an outgoing edge and propagates it back to the self loop.

Example 7. Consider the QUIC graph G



Evaluating the (Self-Loop) rule on Z gives G' on the left. Evaluating the (Self-Prop) rule on Z and $X \cup Y$ pushes the predicate $\nu < 17$ onto the self loop at Z, giving G'' on the right:

$$\frac{}{(G \wedge \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}, B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}} \text{(In-Graph-R)} \qquad \frac{}{(G,B) \vdash \dot{\bigcap} \emptyset \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{\bot}} \text{(Emp)}$$

$$\frac{}{(G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T\Big|_{\top}} \text{(Self-Loop)} \qquad \frac{(G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T\Big|_{B_a \sqcap B_b}} \text{(Self-Prop)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap} T, \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}} \text{(Add-Left)} \qquad \frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} T, \bar{T}^u\Big|_{B_e}} \text{(Add-Right)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} T^i \dot{\subseteq} \dot{\bigcup} T_1^u, \cdots, T_m^u\Big|_{B_0} \qquad (G,B) \vdash \dot{\bigcap} T_j^u \dot{\subseteq} \dot{\bigcup} T_j^u\Big|_{B_j}, \text{ for } j = 1 \cdots m}{(G,B) \vdash \dot{\bigcap} T^i \dot{\subseteq} \dot{\bigcup} T_1^u, \cdots, T_m^u\Big|_{B_0 \sqcap \left(\bigsqcup_{j=1}^m B_j\right)}} \text{(Union-Prop)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} T_j^i \dot{\subseteq} \dot{\bigcup} T_j^i\Big|_{B_j}, \text{ for } j = 1 \cdots m \qquad (G,B) \vdash \dot{\bigcap} T_1^i, \cdots, T_n^i \dot{\subseteq} \dot{\bigcup} T_u\Big|_{B_0}}{(G,B) \vdash \dot{\bigcap} T_1^i, \cdots, T_n^i \dot{\subseteq} \dot{\bigcup} T^u\Big|_{B_0 \sqcap \left(\prod_{j=1}^m B_j\right)}} \text{(Inter-Prop)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i \dot{\subseteq} \dot{\bigcup} T, \bar{T}_a^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} T, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}_b^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}_a^u, \bar{T}_b^u\Big|_{B_a}} \text{(Union-Trans)}$$

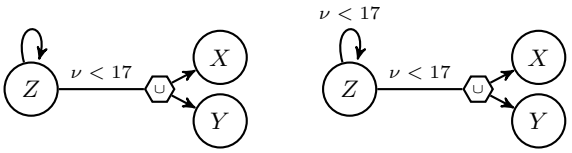
$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i \dot{\subseteq} \dot{\bigcup} T\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} T, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a \sqcap B_b}} \text{(Inter-Trans)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e \sqcap B}} \text{(Base-Str)} \qquad \frac{B \vdash x = y}{(G,B) \vdash \dot{\bigcap} \{x\} \dot{\subseteq} \dot{\bigcup} \{y\}\Big|_{\nu = x}} \text{(Eq-Base)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a \sqcap B_b}} \text{(Double-Edge)}$$

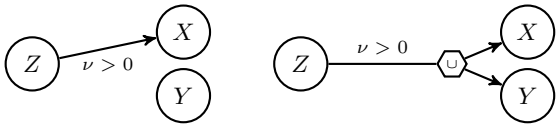
$$\frac{(G,B) \vdash \dot{\bigcap} \{x\} \dot{\subseteq} \dot{\bigcup} \{y\}\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} \{y\} \dot{\subseteq} \dot{\bigcup} \{x\}\Big|_{B_b}}{(G,B) \vdash x = y} \text{(Eq-Set)}$$

Fig. 5. Inference rules for closure of QUIC graphs. **Notation:** $\bar{T}^i, \bar{T}^u$ are sets of vertices, T are individual vertices of the graph, B, B_a, B_b are base abstract states and x, y are base domain variables.



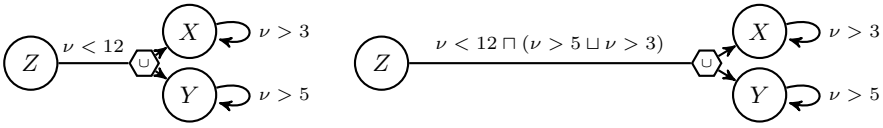
The (Add-Right) rules allows adding extra elements to the union on the right-hand side of an inclusion. (Add-Left) is the dual rule for intersection.

Example 8. Consider the QUIC graph G on the left. Applying (Add-Right) to Z and X, adding Y, gives the QUIC graph G' on the right:



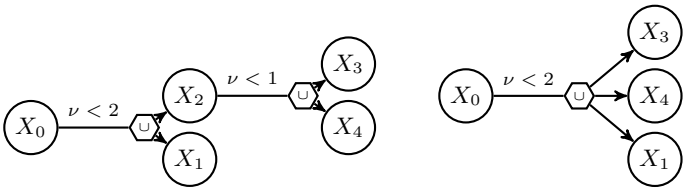
The (Union-Prop) rule pushes information from self loops backward onto edges. (Inter-Prop) performs the same operation on intersections.

Example 9. Consider the QUIC graph G on the left. Applying (Union-Prop) to Z, X and Y yields the graph shown to the right.



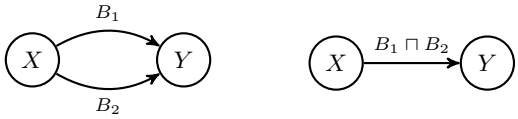
The (Union-Trans) rule combines two union edges to produce a single union edge. This rule loses information from one of the edges, but that information can be regained through the application of (Union-Prop). We write $\sqcap$ and $\sqcup$ for the meet and join operator in the base domain, respectively. (Inter-Trans) does the same for intersection without losing information.

Example 10. Consider the QUIC graph G on the left. The two union edges are combined to produce the union edge on the right. Even though $\nu < 1$ is a stronger constraint than $\nu < 2$, the resulting constraint is the weaker $\nu < 2$.



The (Double-Edge) rule merges two edges between the same vertices into a single edge. QUIC graphs do not track multiple edges between the same two vertices, so a duplicate edge must immediately be converted to a single edge with this rule.

Example 11. Consider the two edges on the left. Since QUIC graphs cannot represent these, they are combined into the single edge on the right.



The rule (Base-Str) strengthens any edge in the graph with the current facts from the base domain B. The rule (Eq-Base) strengthens relationships in the set domain by adding a constraint on the bound variable. The latter also uses equality in the base domain to infer equalities in the set domain. Oppositely, (Eq-Set) uses equalities in the set domain to infer base domain equalities.

Definition 4 (Closure). *Let (G, B) be a QUIC graph and a base domain predicate. The closure (G^*, B^*) is the conjunction of all*

$$\dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e} \text{ such that } (G, B) \vdash \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e}$$

and the constraining of B with all equalities $x = y$ given by the judgment $(G, B) \vdash x = y$.

4.2 Soundness

We first define soundness for systems of inference rules. For a QUIC graph analysis to be sound, the underlying system of inference rules must be sound.

Definition 5 (Inference Soundness). *An inference is sound if the following two conditions hold:*

1. *if $(G, B) \vdash \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e}$, then $\gamma((G, B)) \subseteq \gamma\left(\dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e}\right)$.*
2. *if $(G, B) \vdash x = y$, then for all $(\eta, \eta_B) \in \gamma((G, B))$, we have that $\eta_B(x) = \eta_B(y)$.*

Let us assume that $\widetilde{B}$, the base domain, is a sound abstract domain [5].

Theorem 1. *The inference rules in Figure 5 are sound according to Definition 5.*

4.3 Complexity

Closure of a QUIC graph is potentially expensive since the number of edges in the closure can be exponential in the worst case.

Theorem 2. *There are $O(2^n)$ possible hyperedges in a QUIC graph with n vertices.*

Without "tactics" to apply the rules cleverly in an implementation, the inference over QUIC graphs is intractable.

4.4 Lazy Inference Implementation

We now discuss how the inference operation is implemented in our approach. The goal of the implementation is to avoid a blowup in the number of graph edges and running time each time a closure is to be computed. Lazy inference is a tactic that computes an effective closure on demand. It is composed of many strategies. We describe the most important concepts used in our implementation. (a) *Simplification:* We apply many simplification passes to keep the QUIC graph in a canonical form. This automatically takes into account many of the inference rules from Fig. 5. (b) *Lazy inference:* Instead of computing the closure eagerly and adding a set of extra edges to the graph, we do so lazily whenever edge membership queries are issued by the abstract domain. (c) *Partial closure:* We note that many of the edges generated by a closure are not necessarily useful as invariants for proving properties. Therefore, we have implemented heuristics that choose edges to query. We call this process *candidate generation* since it affects which invariant candidates are considered by our analyzer at each step.

Simplification: Simplification consists of many different parts. The first simplification deals with edges from the empty set $\emptyset$. As such, they do not contribute to the inference. We assume that these edges implicitly exist but do not represent them.

Next, we consider *equivalence classes* of set variables. Two sets X, Y are equivalent if $X \subseteq Y \ \wedge \ Y \subseteq X$. Equivalence classes are identified using a maximal strongly connected component algorithm on the QUIC graph. Equivalence classes of sets can be compacted and one representative is chosen using a pre-defined variable ordering. All membership queries involving members of equivalence classes are first rewritten in terms of the representative members of the classes.

The (Double-Edge) rule is implicitly implemented by our data structure whenever we attempt to add two edges between the same set of nodes. Finally, we use (Self-Loop), (Self-Prop), (Union-Prop) and (Inter-Prop) to propagate labels and add new edges between representatives of equivalence classes. These rules also strengthen the labels on edges.

Lazy Inference: Next, we implement inference on demand by applying the inference rules to decide if a queried edge is present in the graph. This is performed by iterating the (Union-Trans) and (Inter-Trans) rules to compute transitive closures.

Candidate Generation: The computation of a lazy inference is driven by the choice of candidate query edges that we wish to add to the graph. To this end, a candidate generation heuristic is used in our implementation to choose candidate invariant facts. There are many possible heuristics for generating candidate query edges. We use set expressions that appear in the program including properties to be proved as a source of edges to keep in the partial closure. Another

choice includes edges that are generated through transfer functions such as assignments. Once generated, we keep an edge as a candidate edge for future inference computations.

5 Domain Operations

In this section, we will discuss the abstract domain operations over the reduced product domain of QUIC graphs and the base domain $\widetilde{B}$ for base domain variables.

Notation: Let G be a QUIC graph. We will write $G[X \leftarrow X_0]$ to denote the graph obtained by changing the label of vertex X to X_0. We extend the notation to set expressions so that $T[X \leftarrow X_0]$ denotes the substitution of X by X_0 for each occurrence in the expression T.

We define abstract domain transition functions using semantic functions:

$$[\![\text{stmt}]\!]^{\text{S}} : \widetilde{S} \to \widetilde{S}$$

These functions are parameterized by stmt, which is a command in the language of sets and base domain operations. It takes an abstract state $S = (G, B) \in \widetilde{S}$ composed of a graph G and a base domain element B and returns an abstract state S' that represents the state after having executed command stmt on S.

Simple Transfer Functions: The transfer functions for some basic assignment states are represented below. In each case, the result may not be closed. Therefore, we may apply inference on the result, if necessary.

$$[\![\textbf{havoc } \text{X}]\!]^{\text{S}}(G, B) \stackrel{\text{def}}{=} (G[X \leftarrow X_0], B) \qquad X_0 \text{ is fresh}$$

$$[\![\text{X}\textbf{:=}\emptyset]\!]^{\text{S}}(G, B) \stackrel{\text{def}}{=} \left(G[X \leftarrow X_0] \ \wedge \ \begin{array}{c} X \xrightarrow{\ \bot\ } \emptyset \end{array} , B \right) \qquad X_0 \text{ is fresh}$$

$$[\![\text{X}\textbf{:=}\text{T}]\!]^{\text{S}}(G, B) \stackrel{\text{def}}{=} \left(G[X \leftarrow X_0] \ \wedge \ \begin{array}{c} X \rightleftarrows T \end{array} , B \right) \qquad X_0 \text{ is fresh}$$

The command **havoc** X assigns X to a non-deterministic value. Rather than projecting the vertex X from the graph, we rename the existing vertex to a fresh variable X_0. The vertex X_0 remains in the graph as a *history variable*. Operations such as join and widening will eliminate the necessary history variables, ensuring that they do not propagate out of scope. However history variables will exist for as long as possible as this may allow additional relationships to be inferred.

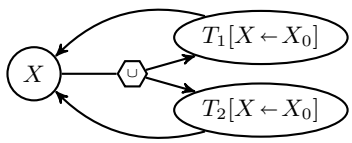
The command X:=$\emptyset$ assigns X to the empty set. Because it is performing a destructive update to X, X is renamed to a history variable X_0 as is standard when performing a destructive update. This leaves the symbol X completely unconstrained so that when constraints are added to X, those are the only constraints on X. The added constraint here is $X \subseteq \emptyset$, labeling it with the strongest possible predicate $\bot$.

The command X:=T assigns a set element T to X. This creates the two edges representing both $X \subseteq T$ and $T \subseteq X$. The edge labels are set to $\top$ and thus not shown as all the information from B can be added to these edges through the inference procedure.

Meet (Intersection): The meet of two abstract states $(G_1, B_1) \sqcap (G_2, B_2)$ is the conjunction of the set constraints and meet in the base domain (i.e., $(G_1 \wedge G_2, B_1 \sqcap B_2)$ where we overload the $\sqcap$ notation for both the QUIC graph and the base domain). Note that viewing the set constraints as graphs, meet is the union of two graphs.

Set Assignment Rule: We now complete domain operations for assignments of the form $X{:=}T_1 \cup T_2 \cdots \cup T_n$ (similarly for $X{:=}T_1 \cap \cdots \cap T_n$). The basic idea is to replace X by a history variable X_0 and introduce hyper-edges to capture the new relations formed. For simplicity, we consider the case $n = 2$

$$[\![\mathtt{X{:=}T_1 \cup T_2}]\!]^{\mathrm{S}}(G, B) \stackrel{\text{def}}{=} \left(G[X \leftarrow X_0] \ \wedge \ \right.$$



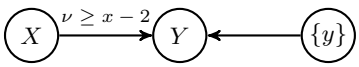
$$\left. , \ B \right)$$

X_0 is fresh

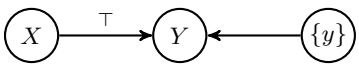
Intersection, disjoint union and set difference operations are similar. They rename X to a fresh variable X_0 and rename T_1 and T_2 similarly, if appropriate. Then a constraint that represents the appropriate equality is added to the QUIC graph.

Base Domain Assignment: An assignment to the base domain variables $x := e$ will result in three changes: (a) applying the assignment to the base domain element B, (b) applying the assignment to each edge label in the QUIC graph G and (c) any singleton node in the graph that involves x needs to be updated either by computing its post-condition w.r.t to the assignment, if invertible or renamed to a fresh set variable X_0 for a destructive assignment. All applications invoke the base domain transfer function and thus rely on the base domain for introduction (or not) of history variables. We illustrate this through a simple example.

Example 12. Consider the QUIC graph G



and let $B : y \geq x$. Consider the destructive assignment $x := y + 1$. The transfer function yields the QUIC graph G':



with the assertion $B' : \ x = y + 1$. We compute a partial closure on the result, which effectively pushes the constraint $x = y + 1$ on the edges of the graph G'.

Choose: The **choose** command selects an element from a set and assigns it to a base domain variable. It takes quantified information from the set domain and applies it to the resulting base domain variable. The strategy to handle $\texttt{x} := \texttt{choose}(T)$ for an abstract state (G, B) is the following:

1. Perform an inference operation on (G, B) giving (G^*, B^*).
2. Extract the base domain constraint B_e from a self-loop on T:
$$G^* = G' \ \wedge \ \dot{\bigcap} T \mathrel{\dot{\subseteq}} \dot{\bigcup} T \Big|_{B_e} \ .$$
3. Replace the bound variable ν in B_e with a fresh base domain variable y giving B_y. This process transfers all the facts that apply to elements in set T and applies them to the variable y.
4. Compute the meet $B' = B^* \sqcap B_y$. This transfers those facts about y to the base domain.
5. Perform the destructive update $\texttt{x} := y$ on (G^*, B') to get the result of choose.

Projection: The projection of a base-domain variable x from (G, B) is performed by (a) projecting x from B and (b) projecting x from each label in G. These are performed by calling the projection defined in the base domain $\widetilde{B}$.

The projection of a vertex T from the QUIC graph G first computes its partial closure (G^*, B^*). Next, we remove all conjuncts involving the vertex T from G^* to obtain the projection.

Join: Let (G_1, B_1) and (G_2, B_2) be the arguments for the join operation. We first compute the partial closure of (G_1^*, B_1^*) of G_1 and likewise the partial closure (G_2^*, B_2^*) of (G_2, B). The join (G, B) is then defined where $B = B_1^* \sqcup B_2^*$ and G is all conjuncts
$$\dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_1 \sqcup B_2}$$
where there exists some G_1' and G_2' such that
$$G_1^* = G_1' \ \wedge \ \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_1} \quad \text{and} \quad G_2^* = G_2' \ \wedge \ \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_2} \ .$$

Widening: As such the QUIC graph domain is a product of a finite graph domain and an abstract base domain. Widening is required iff the base domain does not satisfy the ascending chain condition. The basic widening algorithm is precisely the same as the join operation with the modification that the base domain widening operation is applied for each QUIC edge instead of widening.

6 Evaluation

We now present a preliminary evaluation of our prototype analyzer. The QUIC graphs domain introduced in this paper has two main aspects: (a) it enables relational reasoning between sets to prove that one set (expression) is contained

in another; and (b) it allows us to qualify relations between sets using base domain predicates, in effect allowing us to reason with *set comprehension*. The evaluation in this section is intended to answer the following questions:

1. How much does each of the two ingredients (relations between sets + set comprehensions) add to the ability of the analysis to prove properties of commonly encountered use cases?
2. What is the added cost due to each of the two ingredients to the overall domain?

To carry out the evaluation, we introduce two simplified versions of the QUIC graphs domains namely the 'set' and 'elem' domains. (A) The 'set' domain allows relations between sets but no comprehensions. This is a realization of a container-as-a-whole approach. We create this domain by using the trivial two element $(\bot, \top)$ base domain. (B) The 'elem' domain disallows relations between sets but allows us to reason about the contents of the set using a *summary variable*. This is a realization of a content-centric domain. To simulate this domain, we modify the original QUIC graphs domain to just allow self loops on nodes as the only possible edge. In effect, the predicate on such an edge must be true of every element in the set. Furthermore, the process is exactly equivalent to introducing a *summary variable* for each set variable and performing a base-domain analysis using this summary variable.

Benchmarks: The next step is to choose a series of benchmarks that represent common motifs for set (container) usage in dynamic languages. To evaluate our approach we used two sets of benchmarks. We designed our analysis using the first set of benchmarks, which exercise four commonly occurring operations on containers 'copy', 'filter', 'partition' and 'merge'. We then ran our analysis, *unmodified*, on translated versions of all of the programs from the Python test suite [24] for dictionaries and sets. We removed extraneous parts of these tests and simply translated the core part of each program to an equivalent program in our input language. Each test has a set of pre-defined assertions to be established by our analyzer.

Results: Figure 1 summarizes the results of our analysis run on these benchmarks on an Apple MacBook Pro, on a 2.2GHz Intel Core i7 with 8GB RAM running Mac OS X 10.8.2. We now discuss the comparison of precision and running time. The memory required by most analysis runs was under 150 MB. It is quite clear from the results table that the combination of relational reasoning and comprehension using base domain predicates is quite powerful. Whereas the QUIC graphs domain can prove a majority of the properties, restricting it either by removing the comprehensions (set) or removing the relations between sets (elem) are both able to prove much fewer properties. Furthermore, every property proved by these domains is also proved by the QUIC graphs domain.

The comparison of costs indicates that the QUIC graphs domain is 1.2× slower than the set domain. However, it is 9× slower than the elem domain. The difference in performance is entirely expected since the QUIC graphs domain

Table 1. Results on a set of small benchmarks. **Base Vars**: # of base domain (numerical) variables, **Set Vars**: # of set variables, **Num Prp**: # of assertions to be proved, **T**: Time taken (seconds), **#I**: number of iterations of abstract interpreter before convergence. – represents a time out (600 seconds)

	Base	Set	Num	# Proved			Time Taken (Iterations)					
ID	Vars	Vars	Prp	QG	set	elem	QG_T	(#I)	set_T	(#I)	$elem_T$	(#I)
copy	1	6	2	2	2	0	0.2	(2)	0.2	(2)	0	(2)
filter	4	6	2	2	1	0	0.6	(3)	0.5	(3)	0.1	(2)
generic_max	3	8	6	3	0	0	0.9	(6)	0.6	(6)	0.2	(4)
merge	2	14	2	1	1	0	0.6	(4)	0.6	(4)	0.1	(4)
partition	4	8	4	4	2	0	1.1	(3)	0.9	(3)	0.2	(2)
b_filter	6	6	2	2	0	0	0.7	(3)	0.6	(3)	0.1	(2)
b_map	9	7	2	2	2	2	0.2	(5)	0.3	(5)	0.1	(4)
b_max_min	3	4	1	1	1	1	0.4	(3)	0.3	(3)	0.1	(2)
b_reduce	7	4	1	0	0	0	0.4	(3)	0.3	(3)	0.1	(2)
iter_ind	20	12	1	1	0	0	84.4	(39)	67.9	(39)	6.8	(14)
mul_ret	9	2	2	2	0	0	0.2	(6)	0.1	(6)	0.1	(6)
nest_dep	5	7	1	0	0	0	2.2	(12)	2.2	(12)	0.4	(6)
resize1	15	5	5	4	0	0	1.7	(18)	1.1	(18)	1	(18)
simp_cond	11	5	4	3	0	0	4.6	(12)	1.6	(12)	1.3	(12)
simp_nest	9	10	2	0	0	0	–	(1399)	–	(1612)	0.7	(6)
srange	6	2	2	2	0	0	0.1	(6)	0.1	(6)	0.1	(6)
Total			37	29	9	3	98.3	(125)	77.3	(125)	11.4	(92)

has to perform a lot more reasoning steps. We also find that one example times out (after 600 seconds).

Limitations. While QUIC graphs are an effective abstract domain, but some properties were not proven due to imprecision in the analysis. There are four sources of this imprecision: (1) incomplete candidate generation, (2) imprecise base domain, (3) no cardinality reasoning, and (4) syntactic restrictions within QUIC graphs.

To reduce needless inference in many examples, we use candidate generation (Sect. 4) to reduce the number of rule applications. Because candidate generation reduces the potential edges that can result from a join, it can cause the join to lose more information than is strictly necessary. This is this cause for many of the failures in Table 1, including the failure to prove one of the properties in 'merge'. Further work on candidate generation is quite important.

Because the base domain is also an abstract domain, it is imprecise and may not be able to represent some necessary relationship. This is especially the case when there is a transformation applied to all elements of a set. The base domain must be able to represent that transformation that occurs to each element as a relation. In this test suite there is only one test that exercises this ability and the relations are all representable as linear relationships, so this imprecision does not affect the results. However, if this were a problem, a new base domain could be selected because QUIC graphs are agnostic to the base domain.

The QUIC graphs domain does not track the cardinality of sets. As has been previously shown [18], cardinality can strengthen relationships, and therefore in QUIC graphs, cardinality constraints would create additional closure rules. For example, if for some set X we have that $\{1, 3, 7\} \subseteq X$ and that $|X| = 3$, then we can infer that $X \subseteq \{1, 3, 7\}$. It is possible that cardinality information could provide sufficient information to prove properties that failed in this test suite, but this information could likely be inferred in another way (such as better candidate generation) because most sets in the test suite have unknown cardinality.

QUIC graphs are syntactically restricted to allow comprehensions only on one side of a subset relationship. Reverse inclusions (Sect. 3) are not supported. We hypothesize that the ability to know that an element exists in a set will be beneficial when abstracting other containers using sets.

7 Related Work

There exists a large number of container analyses, mostly focused on arrays. Although there are many different approaches, the problem is fundamentally the same: partitioning an array in order to summarize different segments. Gopan et al. [13], Halbwachs et al. [15] and Cousot et al. [7] use an abstract interpretation framework with materialization and summarization. Therein, the partitions are inferred from the structure of the program. Seghir et al. [25] perform this in the context of predicate abstraction, similarly to abstract interpretation. Jhala et al [16], McMillan [22], Kovacs et al. [17], and Dillig et al. [9, 10] use theorem provers to perform this partitioning. Our approach does not use a partitioning scheme except for the special case of loops that iterate over sets. Furthermore, these approaches do not, in general, reason about comprehensions or relate the contents of different arrays.

There are several alternative approaches to reasoning about container manipulations. Marron et al. [20, 21] used a shape analysis to emulate data storage of containers. They used appropriate inductive predicates with carefully tuned, simplified implementations of the containers to get an automatic analysis. Dillig et al. [11] extended their previous work on arrays to more generic containers. Their approach uses base domain predicates as constraints on the sets of keys for maps. This is a highly tuned example of what we have been calling a content-centric domain. Their approach does not directly infer relationships between containers. However, they can indirectly infer relations through data invariants that relate their contents. Finally, Pham et al. [23] introduced a relational domain for sets. Their domain is similar to ours in that it is designed to directly represent relations between sets. Their approach represents what we term an as-a-whole approach for the most part. It does support a base domain of uninterpreted functions and can be precise for a restricted class of programs. Because they support only uninterpreted functions for the base domain, they have been able to implement some under-approximations required to infer equalities with predicate comprehensions, but this base domain does not support any manipulation or reductions and thus is weaker than domains that we support.

The invariant generation procedure of [14] could infer many of the invariants that we infer given a sufficiently expressive list of predicate templates. They select from templates to use for quantified facts. As a result, their analysis requires user input and guidance for success, but the approach does offer some additional generality. Bouajjani et al. [3] present a similar, more automatic approach to dealing with quantified invariants, by pre-selecting appropriate templates for many applications. They apply their work to linked list structures and support multiple bound variables to be able to maintain sortedness properties. Like the work of [20], they use a shape analysis framework to approximate the shape and data of lists, while maintaining quantified side conditions on an integer base domain.

The QUIC graph data structure is similar to a formalization of constraint graphs [2, 12] use to prove complexity of satisfaction of constraints [1]. While the encoding is similar, there is no need for base domain labels since constraint graphs are unable to place quantified restrictions on the contents of the sets they constrain. In general, constraint graphs do represent sets, but they are intended to use sets to analyze programs rather than analyzing set-manipulating programs.

The decision procedures community has largely solved this problem of relational containers, but only for the problem of entailment checking. Decision procedures do not perform invariant generation. Bradley et al. [4] demonstrated decision procedures for arrays and other containers. The Z3 SMT solver implements an optimized version [8] of these decision procedures to speed up these problems. Also, Lam et al. [19] and Kuncak [18] developed a system that simultaneously reasons about sets and their cardinalities relationally. Since these tools solve the decision problem rather than the inference problem, they are incomparable, however the optimizations used in [8] are similar to operations that we define in our closure because they are Boolean algebra-like operations.

8 Conclusion

We have demonstrated a relational abstract domain for sets that combines a content-centric analysis with a container-as-a-whole approach. This is achieved through a new representation for set constraints called QUIC graphs that simplifies the representation of set expressions and inclusion relations that use comprehensions. Our evaluation of this domain shows that a combined approach using QUIC graphs is quite effective in practice. It outperforms weaker alternatives such as a content-centric approach and a container-as-a-whole approach.

Going forward, we are developing tighter integration of our domain to analyze a range of data structures in dynamic languages such as Python and JavaScript. Our future work will complete the QUIC graph structure to encode *reverse inclusion relations* (see Section 3), track set cardinalities more effectively and enable the tracking of auxiliary data that can help extend this analysis to specific structures such as arrays, lists and dictionaries.

Acknowledgments. We would like to thank Xavier Rival and the CUPLV group for insightful discussions on this work, as well as the anonymous reviewers for the helpful comments. This work is supported in part by the National Science Foundation through grants CCF-1055066 and CCF-1218208.

References

[1] Aiken, A., Kozen, D., Vardi, M., Wimmers, E.: The complexity of set constraints. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 832, pp. 1–17. Springer, Heidelberg (1995)

[2] Aiken, A., Fähndrich, M., Foster, J.S., Su, Z.: A toolkit for constructing type- and constraint-based program analyses. In: Leroy, X., Ohori, A. (eds.) TIC 1998. LNCS, vol. 1473, pp. 78–96. Springer, Heidelberg (1998)

[3] Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)

[4] Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)

[5] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)

[6] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL (1979)

[7] Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL (2011)

[8] de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Conference on Formal Methods in Computer Aided Design, FMCAD (2009)

[9] Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. Weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010a)

[10] Dillig, I., Dillig, T., Aiken, A.: Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In: OOPSLA (2010b)

[11] Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. In: POPL (2011)

[12] Flanagan, C.: Effective Static Debugging via Componential Set-Based Analysis. PhD thesis, Rice University (1997)

[13] Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: POPL (2005)

[14] Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL (2008)

[15] Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI (2008)

[16] Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)

[17] Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)

[18] Kuncak, V.: Modular Data Structure Verification. PhD thesis, EECS Department, Massachusetts Institute of Technology (2007)
[19] Lam, P., Kuncak, V., Rinard, M.: Hob: a tool for verifying data structure consistency. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 237–241. Springer, Heidelberg (2005)
[20] Marron, M., Stefanovic, D., Hermenegildo, M., Kapur, D.: Heap analysis in the presence of collection libraries. In: PASTE (2007)
[21] Marron, M., Méndez-Lojo, M., Hermenegildo, M., Stefanovic, D., Kapur, D.: Sharing analysis of arrays, collections, and recursive structures. In: PASTE (2008)
[22] McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
[23] Pham, T.-H., Trinh, M.-T., Truong, A.-H., Chin, W.-N.: FixBag: A fixpoint calculator for quantified bag constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 656–662. Springer, Heidelberg (2011)
[24] Python. Python 2.7.3 test suite (2012), `http://www.python.org`
[25] Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)

Reducing Lookups for Invariant Checking

Jakob G. Thomsen[1,⋆], Christian Clausen[1], Kristoffer J. Andersen[1], John Danaher[2], and Erik Ernst[1,⋆⋆]

[1] Aarhus University
{gedefar,christia,kja,eernst}@cs.au.dk
[2] Google Inc.
jsd@google.com

Abstract. This paper helps reduce the cost of invariant checking in cases where access to data is expensive. Assume that a set of variables satisfy a given invariant and a request is received to update a subset of them. We reduce the set of variables to inspect, in order to verify that the invariant is still satisfied. We present a formal model of this scenario, based on a simple query language for the expression of invariants that covers the core of a realistic query language. We present an algorithm which simplifies a representation of the invariant, along with a mechanically verified proof of correctness. We also investigate the underlying invariant checking problem in general and show that it is co-NP hard, i.e., that solutions must be approximations to remain tractable. We have seen a factor of thirty performance improvement using this algorithm in a case study.

Keywords: Invariant checking, CNF reductions, co-NP hard.

1 Introduction

An invariant is a useful and well-known device for specifying data consistency. Assuming that a given amount of data satisfies a specific invariant, the associated notion of consistency may be preserved across an update by checking that the invariant holds in the updated data, and rejecting the update if it does not hold. However, invariant checking may refer to unmodified data as well as updated data, and some unmodified data may well be stored in such a way that access is expensive, e.g., on a remote server, or even impossible, e.g., if the update is performed on a mobile device that currently has no network access. This paper presents a provably correct algorithm for simplifying the invariant to a form that uses a smaller amount of data and still correctly determines whether the invariant has been violated. As a result, consistency checking may now be performed using fewer resources, or even in some situations where, otherwise, it could not be performed at all.

⋆ Part of work done while employed at Google Inc.

⋆⋆ Supported by Google Faculty Research Awards 2012. Note that this paper could not participate in the artifact evaluation process, because Erik was a co-chair of the Artifact Evaluation Committee.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 426–450, 2013.

We provide a model of this scenario, with a simple query language for expressing invariants and an algorithm that simplifies invariants. Data is modeled as a finite set of variables V with values $D : V \to \mathsf{Value}$, where Value is some domain of values modeling the results of expression evaluation; a subset $V' \subseteq V$ is modified to have new values $U : V' \to \mathsf{Value}$, extended to cover all variables by setting $U(v) = D(v)$ for $v \in V \backslash V'$. Finally, a notion of consistency among the variables is specified by means of an invariant $I : (V \to \mathsf{Value}) \to \mathsf{Boolean}$, given as an expression in the query language. For instance, the invariant could specify that two variables have the same value, that one is greater than the other, etc. However, we immediately reduce the Value domain to $\mathsf{Boolean}$, leaving general (and undecidable) expression evaluation implicit, and representing the expressions at the logical level simply as $\mathsf{Boolean}$ variables. As a consequence of this, each variable in the formal model corresponds to an entire expression (without side-effects) in the invariants of our case study. Moreover, the case study deals with simple objects whose fields may hold references to other objects or values of primitive types, and each invariant is associated with specific object types, such that it is applicable to a given update iff it affects objects of those types. Still, it is useful to study our simplified formal model because it preserves and thus highlights a difficult (co-NP hard) core problem.

Our algorithm analyzes and simplifies the invariant, thus detecting opportunities for reducing the cost of checking the invariant. We validate the algorithm by a proof of correctness. Finally, we show that the underlying problem is co-NP hard. This means that a direct solution is intractable, which justifies that our algorithm is sound but not complete, i.e., the simplified invariant will correctly determine whether or not a violation of the original invariant exists, but the reduced set of variables may not be minimal. In summary, the contributions of this work are as follows:

- Specifying an invariant model and an algorithm that safely simplifies such an invariant, thereby reducing the amount of data that needs to be checked to detect invariant violations caused by an update
- Providing a mechanically verified correctness proof in Coq, stating that our algorithm is sound
- Proving that the underlying problem is co-NP hard, thus establishing that it is intractable to obtain a perfect (complete) solution
- Discussing a number of possible extensions and generalizations, thus clarifying the design space in which tractable approximations to the perfect solution must be explored
- Presenting a case study of the algorithm, where it decreases the number of lookups by a factor of thirty compared to a unoptimized algorithm, and it is approximately equal (within 3.5%) to an algorithm where the treatment of each invariant is manually optimized.

The rest of the paper is structured as follows: In Sect. 2 we informally present the ideas in the algorithm. Section 3 presents our formal model, Sect. 4 presents the correctness theorem, and the complexity of the underlying problem is discussed in Sect. 5. Section 6 reports on a case study where the algorithm was used

in practice. Section 7 discusses the algorithm and results and some interesting extensions. Finally, Sect. 8 discusses related work and Sect. 9 concludes.

2 Example

Before we go into the formal details, this section gives the intuition behind our algorithm, based on an example execution of the algorithm.

Our algorithm takes as input an invariant of a data store and an update to that data store. With the reduction of the Value domain to Boolean that we mentioned in the introduction, the invariant is simply a Boolean expression.

Eq. (1) shows the invariant (inspired by our case study) that we will use in our running example, and Eq. (2) shows the update.

$$(cn \vee (cn \wedge ln)) \wedge ((top \vee area) \Rightarrow (cref \wedge ln)) \wedge (top \vee cref) \tag{1}$$

$$[area \mapsto \texttt{false}, ln \mapsto \texttt{false}] \tag{2}$$

We use standard logical notation: $\Rightarrow$, $\wedge$, $\vee$, and $\neg$ for implication, conjunction, disjunction, and negation, respectively. The set of logical operators could easily be extended as long as each operator can be reduced by simple (polynomial) rewriting schemes; our formal model uses only conjunction, disjunction, and negation. We represent an update as a map from variables to boolean values (`true` or `false`). In the example, both *area* and *ln* are updated to `false`. Note that 'updated' does not necessarily mean changed — they might have been `false` before the update as well. Given the invariant and update, our algorithm will simplify the invariant to the single literal shown in Eq. (8).

The invariant is used to verify the relationship between a mountain and its enclosing country. The interpretation of the variables is as follows: *cn* signifies that a given mountain has a common name; *top* signifies that the top of the mountain is within the border of a given country; *area* denotes that at least a certain percentage of its area is within the border of a given country; *cref* signifies that the mountain has a reference to its enclosing country; and finally *ln* signifies that the mountain has a local name specific to the country. The update then signifies that for a given mountain and country, less than a certain percentage of the mountain's area is within the country, and that the mountain does not have a local name. In reality, an arbitrary amount of computation may be needed to determine these values, but we abstract that away and look only at the result of the computations in a logical setting. The case study in Sect. 6 elaborates more on this.

For completeness we also give an example of a data store. It is not needed by our algorithm which only depends on the update and the invariant, but it is of course crucial for specifying what correctness means. We represent a data store similarly to an update, namely as a map from variable names to boolean values. In Eq. (3) we show a possible data store that satisfies the invariant.

$$\begin{aligned}[cn \mapsto \texttt{true}, top \mapsto \texttt{false}, area \mapsto \texttt{true},\\ cref \mapsto \texttt{true}, ln \mapsto \texttt{true}]\end{aligned} \tag{3}$$

The update should only be committed to the data store if the data store will satisfy the invariant after the commit. Hence, a naive way of checking the invariant would be to compute its value using the update and the data store as needed, and only commit the update if the invariant is satisfied. Our focus is to reduce the costs of such a naive approach based on the assumption that variable lookup is expensive. We do this by simplifying the invariant while maintaining soundness: Invariant violation will be faithfully detected using the simplified invariant. We do this by incorporating the information from the update, and we utilize that the invariant was satisfied for the data store before the update.

Note that we cannot avoid the costly variable lookups by caching the old values of all variables in the invariant, which could otherwise entirely eliminate the need for those costly variable lookups for subsequent updates, and thus eliminate the need for our solution for all updates except the first one.

Our concrete case study illustrates one way in which this type of caching is made impossible. The caching of variables of the formal model corresponds to the caching of results of expression evaluation involving object fields in the case study, and the invariants are chosen according to the types of the objects. For example, we may have an invariant stating that if a house H has a zipcode N and is located in a city C, then N must be the zipcode of a part of C. The variables involved would be the fields of particular objects of type house and city etc. containing some updated values, and we would essentially have to cache all such objects in order to be able to cache the old values of all variables for all usages of this particular invariant. Since the number of objects of any given type could easily be very large, this already rules out caching of the old values of variables. With many invariants and many types of objects, it just becomes even more impossible.

We could have chosen to work with arrays in the formal model rather than single variables, in order to replicate the core of this phenomenon. An invariant would then specify a required relation among the variables in the k'th entry of the arrays (so we would consider, e.g., $top[k] \vee cref[k]$ rather than simply $top \vee cref$). In this setting, the choice of k corresponds to the selection of a set of objects to update and check in the case study. The fact that k changes in unpredictable ways for each update shows that we cannot cache the old values (here: $top[k]$ and $cref[k]$), because we would need to do that for every k in order to ensure that the relevant old values for the next update are in the cache.

However, we do not use arrays like this in our formal model, because they are just one possible reason why caching of the old values of variables may be impossible; typed objects and type specific invariants is another such reason, and there may be many others, each with its own particular structure. We have instead chosen to make it an *assumption* that this type of caching is impossible. Consequently, our model is faithful to the intended semantics, but avoids a significant amount of complexity, and remains directly applicable in all cases where this type of caching is ruled out for whatever reason.

Before the algorithm starts to simplify the invariant, it is converted into conjunctive normal form (CNF), i.e., using only conjunction, disjunction, and

negation, with negations only at the leaves and disjunctions nested inside conjunctions. From the invariant in Eq. (1) the corresponding CNF is shown in Eq. (4). Many variables may be duplicated during this transformation, but in return the shared variables enable many simplification steps. Conversion to CNF is in the worst case an exponential blowup in the size of the expression, but in practice it is usually not a problem, as an invariant is typically written as a big conjunction of smaller constraints. Another strong hint that this transformation is acceptable is the fact that SAT solvers [13] often start by transforming their input to CNF.

$$\begin{gathered}(cn) \wedge (cn \vee ln) \\ \wedge \\ (top \vee cref) \\ \wedge \\ (\neg top \vee cref) \ \wedge \ (\neg top \vee ln) \wedge (\neg area \vee cref) \ \wedge \ (\neg area \vee ln)\end{gathered} \tag{4}$$

Our algorithm simplifies the invariant in four phases. For the first phase, the algorithm uses the property that the data store satisfies the invariant before the update. The algorithm does so by searching for clauses with a single non-updated literal. Note that we adopt the terminology from SAT solvers and call clauses with one literal for unit clauses [26]. In Eq. (4) (cn) is such a non-updated unit clause. Since the data store satisfies the invariant, we know that the unit clause (cn) must have the value `true`, otherwise the invariant would not be satisfied. Using this information about (cn) we know that the literal cn is `true` and that $\neg cn$ is `false`. Hence, clauses containing cn are `true` and does not need to be simplified further, and $\neg cn$ can be removed from clauses containing it. We can perform this transformation even though some of the affected clauses contain updated variables, because they cannot influence the value of the non-updated variables. The first phase is repeated until a fixed point is reached. For our example, the first phase finds one suitable literal cn, and the algorithm simplifies Eq. (4) to Eq. (5).

$$\begin{gathered}(top \vee cref) \\ \wedge \\ (\neg top \vee cref) \ \wedge \ (\neg top \vee ln) \wedge (\neg area \vee cref) \ \wedge \ (\neg area \vee ln)\end{gathered} \tag{5}$$

In the second phase, the algorithm inserts values from the update into the formula. Since *area* is updated to `false`, we know that $\neg area$ is `true` and hence the clauses containing it are true and may be eliminated. Similarly, as ln is updated to `false` we remove it from its clause. Equation (6) shows the simplified version of the invariant, after removing the satisfied clauses.

$$\begin{gathered}(top \vee cref) \\ \wedge \\ (\neg top \vee cref) \wedge (\neg top)\end{gathered} \tag{6}$$

Note that $(\neg top)$ cannot be eliminated since it is a unit clause that was produced by simplifying a clause containing updated variables. This means that we do not know which of the literals in the original (non-unit) clause that satisfied the clause, so we cannot infer the value of any variable from such a unit clause, and in particular we cannot infer the value of *top*. Consequently, this clause must be preserved so that we remember to check that *top* is `false` in the updated data store, because the invariant would otherwise be violated.

In the third phase we eliminate clauses that contain all of the literals in another clause. In Eq. (6), $(\neg top \vee cref)$ and $(\neg top)$ are such a pair, where we therefore eliminate $(\neg top \vee cref)$. The reason why this transformation is sound is that if $(\neg top)$ is `true` then so is $(\neg top \vee cref)$. If on the other hand $(\neg top)$ is `false`, the entire expression is `false`. Therefore the truth value of $(\neg top \vee cref)$ does not matter. The resulting invariant is shown in Eq. (7).

$$\begin{gathered}(top \vee cref) \\ \wedge \\ (\neg top)\end{gathered} \tag{7}$$

Note that in the third phase, the check is based on literals and not variables, so $(\neg top)$ is contained by $(\neg top \vee cref)$, but not by $(top \vee cref)$, and hence only the former can be eliminated.

In the fourth phase the algorithm again uses the property that the invariant was satisfied before committing the update. We therefore know that all clauses were `true` before the update, and clauses that never contained any updated literals will still be true after the update. Hence the algorithm is free to eliminate clauses that initially did not contain updated literals. In the example, the algorithm eliminates the clause $(top \vee cref)$ from Eq. (6), which simplifies the invariant to the single literal in Eq. (8).

$$(\neg top) \tag{8}$$

At this point our algorithm cannot simplify the invariant anymore. To detect a potential violation we therefore look up *top* in the data store. In this case it is `false`, and hence the invariant is still satisfied.

3 The Invariant Simplification Algorithm

This section presents the query language for expressing invariants and the algorithm, along with correctness theorems.

Figure 1(a) shows the syntax of the query language used to express invariants, which is simply propositional logic. We assume that the invariant has already been converted to CNF and hence the grammar only admits CNF expressions. Borrowing the terminology from SAT solvers [13] we denote a variable x as a *positive literal*, a negated variable $\neg x$ as a *negative literal*, and a topmost disjunction as a *clause*. The semantics is presented in Fig. 1(b) where U represents the update and D represents the data store, both mapping variables to `true`

(a) Query language syntax:

$$\begin{aligned}\mathbf{b} &::= \mathbf{b} \wedge \mathbf{b} \mid b\\ b &::= b \vee b \mid l\\ l &::= x \mid \neg x\end{aligned}$$

(c) Set representation of invariants:

$$\begin{aligned}I &::= [\,\overline{(C,d)}\,]\\ C &::= \mathcal{L} \mid \top\\ \mathcal{L} &::= 2^l\end{aligned}$$

(b) Query language semantics:

$$lookup(U, D, x) = \begin{cases} U[x] & \text{if } x \in dom(U) \\ D[x] & \text{if } x \in dom(D) \backslash dom(U) \end{cases}$$

$$\text{E-PosLit}\ \frac{lookup(U, D, x) = \texttt{true}}{U, D \vdash x} \qquad \text{E-NegLit}\ \frac{lookup(U, D, x) = \texttt{false}}{U, D \vdash \neg x}$$

$$\text{E-Disj}\ \frac{\exists i \in \{1, 2\} : \quad U, D \vdash b_i}{U, D \vdash b_1 \vee b_2} \qquad \text{E-Conj}\ \frac{U, D \vdash \mathbf{b}_1 \quad U, D \vdash \mathbf{b}_2}{U, D \vdash \mathbf{b}_1 \wedge \mathbf{b}_2}$$

(d) Set representation transformation:

$$\begin{aligned}\mathcal{S}[\![b]\!](U) &= [\mathcal{S}'[\![b]\!](U)]\\ \mathcal{S}[\![\mathbf{b}_1 \wedge \mathbf{b}_2]\!](U) &= \mathcal{S}[\![\mathbf{b}_1]\!](U) +\!+ \mathcal{S}[\![\mathbf{b}_2]\!](U)\\ \mathcal{S}'[\![x]\!](U) &= (\,\{x\},\ x \in dom(U)\,)\\ \mathcal{S}'[\![\neg x]\!](U) &= (\,\{\neg x\},\ x \in dom(U)\,)\\ \mathcal{S}'[\![b_1 \vee b_2]\!](U) &= (C_1 \cup C_2, d_1 \vee d_2)\\ &\quad \text{where}\\ &\quad (C_i, d_i) = \mathcal{S}'[\![b_i]\!](U)\end{aligned}$$

(e) Auxiliary functions:

$$\begin{aligned}pLits(C) &= \{\ x \mid x \in C\ \}\\ nLits(C) &= \{\ x \mid \neg x \in C\ \}\\ Vars(C) &= pLits(C) \cup nLits(C)\\ pLits(I) &= \textstyle\bigcup_{(C,\_) \in I} pLits(C)\\ nLits(I) &= \textstyle\bigcup_{(C,\_) \in I} nLits(C)\\ Vars(I) &= pLits(I) \cup nLits(I)\end{aligned}$$

(f) Set representation semantics:

$$\text{S-Pos}\ \frac{x \in pLits(C) \qquad lookup(U, D, x) = \texttt{true}}{U, D \vdash C}$$

$$\text{S-Neg}\ \frac{x \in nLits(C) \qquad lookup(U, D, x) = \texttt{false}}{U, D \vdash C}$$

$$\text{S-Trivial}\ \ U, D \vdash \top \qquad \text{S-Nil}\ \ U, D \vdash [\,] \qquad \text{S-Cons}\ \frac{U, D \vdash C \qquad U, D \vdash I}{U, D \vdash (C, \_) :: I}$$

Fig. 1. Definitions regarding invariants

or `false`. The judgment $U, D \vdash \mathbf{b}$ signifies that the expression $\mathbf{b}$ is *satisfied* under the update U and data store D. The rules are straightforward, noting that lookup consults both U and D, giving priority to U.

The algorithm works on a set representation, defined in Fig. 1(c), rather than directly on a query language expression. A clause C is a set of literals $\mathcal{L}$ or the special value $\top$, which signifies that the clause is trivially satisfied. An invariant I is a list of clause and *dirty bit* pairs. The dirty bit is denoted d. In spite of the fact that $\top$ is trivially satisfied and the empty disjunction is not, it turns out to be convenient to treat $\top$ as an empty set with the usual set operators such as $\in, \cup, \cap$. We use a terminology similar to the query language and refer to *literals*, *clauses* (sets of literals), and *invariants* (lists of clauses).

The transformation from the query language to the set representation is given in Fig. 1(d) as the function $\mathcal{S}$. The transformation recursively traverses the query and collects all literals from each clause into a set, and all such sets into a list. The intuition is that there is a conjunction between the sets and there is a disjunction between the literals in each set. Notationally, '::' constructs a list from an element and a list, '++' appends two lists, and $[\_]$ creates a list by enumeration of its elements. The clause representation also carries information about whether the clause initially contained an updated variable represented by the dirty bit. It is `true` iff the clause contained an updated variable. The dirty bit is crucial because some transformations require this bit to be `false` for soundness, and it cannot be inferred from the clauses themselves.

Figure 1(e) shows some auxiliary syntactic functions. $pLits(C)$ and $pLits(I)$ return the positive variables of a clause or invariant, and $nLits(\_)$ is similar but returns the negative variables. $Vars(C)$ and $Vars(I)$ return all the variables in a clause and an invariant, respectively. Finally, Fig. 1(f) defines the semantics of the set representation. The underscore '_' denotes a value that does not need a name because it is not used elsewhere.

The following lemma shows that $\mathcal{S}$ preserves the satisfiability of an invariant, i.e., that the set representation is faithful (the proof is in the Coq code):

Lemma 1 (Correctness of $\mathcal{S}$). $U, D \vdash \mathbf{b} \iff U, D \vdash \mathcal{S}[\![\mathbf{b}]\!](U)$

The dirty bit must initially be set correctly—if it is `false` then the set of literals do not contain an updated literal. The converse is not required for correctness, but quality of the output is reduced if it is violated. The wellformedness property is defined in Def. 1:

Definition 1 (Wellformed invariant). *Given I, U, and D, the invariant I is* well-formed with respect to U and D, *written as $wf(I, U, D)$, if and only if*

$$\forall (C, \texttt{false}) \in I \,.\, C \neq \top \Rightarrow Vars(C) \cap dom(U) = \emptyset \;\wedge\; Vars(I) \subseteq dom(D).$$

We need two auxiliary functions in the algorithm. The function $\mathcal{R}$ is shown in Fig. 2(a). Given a variable and a boolean value, $\mathcal{R}$ returns the simplified invariant where the variable has been replaced by the given value. If the boolean value is `true` then all clauses containing positive occurrences of the variable are collapsed to $\top$, whereas all negative occurrences are removed without changing

(a) Reducing a set given the value of one variable; finding unit clauses:

$$\mathcal{R}[\![C]\!](x,\mathtt{b}) = \begin{cases} C & \text{if } x \notin C \text{ and } \neg x \notin C \\ \top & \text{if } x \in C \text{ and } \mathtt{b} = \mathtt{true} \text{ or } \neg x \in C \text{ and } \mathtt{b} = \mathtt{false} \\ C \setminus \{\, x \,\} & \text{if } x \in C \text{ and } \mathtt{b} = \mathtt{false} \text{ and } \neg x \notin C \\ C \setminus \{\, \neg x \,\} & \text{if } \neg x \in C \text{ and } \mathtt{b} = \mathtt{true} \text{ and } x \notin C \end{cases}$$

$$\mathcal{R}[\![I]\!](x,\mathtt{b}) = \begin{cases} [\,] & \text{if } I = [\,] \\ (\mathcal{R}[\![C]\!](x,\mathtt{b}),d) :: \mathcal{R}[\![I']\!](x,\mathtt{b}) & \text{if } I = (C,d) :: I' \end{cases}$$

$$\mathcal{F}_u[\![I]\!] = \begin{cases} \text{SOME } C \text{ if } I = I' {+\!+} (C,\mathtt{false}) :: I'', \\ \qquad \text{where } |C| = 1 \text{ and } |C'| \neq 1 \text{ for all } (C',\mathtt{false}) \in I' \\ \text{NONE} \quad \text{otherwise} \end{cases}$$

(b) Phase one, utilizing unit clauses:

$$\mathcal{P}_1[\![I]\!] = \begin{cases} I & \text{if } \mathcal{F}_u[\![I]\!] = \text{NONE} \\ \mathcal{P}_1[\![I']\!] & \text{where SOME } C = \mathcal{F}_u[\![I]\!] \text{ and } C = \{x\} \text{ for some } x \text{ and} \\ & \quad I' = \mathcal{R}[\![I]\!](x,\mathtt{true}) \\ \mathcal{P}_1[\![I']\!] & \text{where SOME } C = \mathcal{F}_u[\![I]\!] \text{ and} C = \{\neg x\} \text{ for some } x \text{ and} \\ & \quad I' = \mathcal{R}[\![I]\!](x,\mathtt{false}) \end{cases}$$

(c) Phase two, inserting values from the update:

$$\mathcal{P}_2[\![I]\!](U) = \begin{cases} I & \text{if } U = [\,] \\ \mathcal{P}_2[\![I']\!](U') & \text{where } U = (x,\mathtt{b}) :: U' \text{ and } I' = \mathcal{R}[\![I]\!](x,\mathtt{b}) \end{cases}$$

(d) Phase three, eliminating every clause for which there is a subset clause:

$$\mathcal{P}_3[\![I]\!] = \begin{cases} \mathcal{P}_3[\![I']\!] & \text{if } I = I_1 {+\!+} (\mathcal{L},d) :: I_2, I' = I_1' {+\!+} (\mathcal{L},d) :: I_2' \\ & \quad I_1' = \mathcal{P'}_3[\![\mathcal{L},I_1]\!],\ I_2' = \mathcal{P'}_3[\![\mathcal{L},I_2]\!], \\ & \quad \exists (\mathcal{L}',\_) \in I_1 {+\!+} I_2 \,.\, \mathcal{L} \subseteq \mathcal{L}', \text{ where } I_1 \text{ is as short as possible} \\ I & \text{if the above constraints cannot be satisfied} \end{cases}$$

$$\mathcal{P'}_3[\![\mathcal{L},I]\!] = \begin{cases} [\,] & \text{if } I = [\,] \\ (\top,d) :: \mathcal{P'}_3[\![\mathcal{L},I']\!] & \text{if } I = (\mathcal{L}',d) :: I' \text{ and } \mathcal{L} \subseteq \mathcal{L}' \\ (\mathcal{L}',d) :: \mathcal{P'}_3[\![\mathcal{L},I']\!] & \text{if } I = (\mathcal{L}',d) :: I' \text{ and } \mathcal{L} \not\subseteq \mathcal{L}' \\ (\top,d) :: \mathcal{P'}_3[\![\mathcal{L},I']\!] & \text{if } I = (\top,d) :: I' \end{cases}$$

(e) Phase four, eliminating all non-dirty clauses:

$$\mathcal{P}_4[\![I]\!] = \begin{cases} [\,] & \text{if } I = [\,] \\ (\top,\mathtt{false}) :: \mathcal{P}_4[\![I']\!] & \text{if } I = (C,\mathtt{false}) :: I' \\ (C,\mathtt{true}) :: \mathcal{P}_4[\![I']\!] & \text{if } I = (C,\mathtt{true}) :: I' \end{cases}$$

Fig. 2. The invariant reduction algorithm

the truth value of the containing clause. The converse is done if the boolean value is `false`. The function $\mathcal{F}_u$ finds unit clauses and is shown in Fig. 2(a), too.

The first phase of the algorithm, shown as the function $\mathcal{P}_1$ in Fig. 2(b), iteratively finds non-updated unit clauses and utilizes that the unit clause was satisfied before the update and hence is also satisfied after, as it did not contain an updated literal. Therefore any clauses containing the literal in the unit clause, which we will call the unit literal, are also true. Furthermore the negated version of the unit literal can be removed from any clauses containing it, as the negated literal is always `false`. This process is done iteratively until there are no more unit clauses. In the definition of $\mathcal{P}_1$ we use option types to distinguish between the two states where there was a unit clause (SOME C) and there was not (NONE).

In the second phase of the algorithm, $\mathcal{P}_2$ in Fig. 2(c), the invariant is simplified with respect to the values in the update.

During phase three, $\mathcal{P}_3$ in Fig. 2(d), we utilize the subset relation between clauses. Given an invariant, I, and a clause, $\mathcal{L}$, this phase transforms all the clauses that are supersets of $\mathcal{L}$ to $\top$. This transformation is useful because removing the bigger clause could reduce the number of variables to look up in the data store. The transformation is provably sound, but it is worth considering the intuition as well. If $\mathcal{L}$ is satisfied, so are all clauses containing $\mathcal{L}$. Violation of I due to any of these clauses will then definitely make $\mathcal{L}$ `false` and hence be revealed as long as $\mathcal{L}$ is present. We do not know for sure that any of the eliminated clauses are `false` when $\mathcal{L}$ is `false` (each of them could be `true` due to some additional literal), but that is unimportant because the invariant is in any case violated. We require that I_1 be chosen such that no shorter value for I_1 satisfies the constraints; this is simply needed in order to make sure that $\mathcal{P}_3$ is a function rather than a relation. The implementation of this phase uses a number of techniques to obtain good performance, e.g., working on an invariant whose clauses are sorted by size, but for the presentation here we have given priority to readability because these techniques are generic and well-known.

In contrast to phase one, the third phase does not depend on the dirty bit. We still need the variables in the preserved clause to reverify the invariant. In $\mathcal{P}_1$ we also utilize information about the negated version of the unit literal to remove literals from other clauses, so $\mathcal{P}_1$ is not just a special case of $\mathcal{P}_3$.

Phase four, $\mathcal{P}_4$ in Fig. 2(e), eliminates certain clauses because they are known to be satisfied (whether or not earlier phases have modified them). The intuition is that every non-dirty clause will be `true` after the update because it was `true` before the update and it contains no updated variables.

If a clause is simplified to the empty clause, it is not satisfiable. We may therefore check for this situation, as a small optimization, to stop and report the invariant violation immediately.

We can see as follows that the phases of the algorithm are ordered optimally:

- The first phase is iterated until no progress is made. This is useful because each step may produce new non-dirty unit clauses by removing literals from

non-dirty sets. The third phase is iterated to use every subset relation, but the transformation itself does not create subset relations.
- No other phase produces opportunities for running the same phase or any other phase again: $\mathcal{P}_2$ depends on the values of updated variables, and no extra updated variables are ever introduced; $\mathcal{P}_3$ depends on subset relations, but phase one either reduces clauses to $\top$ or removes the same literals from all clauses, which never changes any subset relations; similarly for phase two. Finally, $\mathcal{P}_4$ removes non-dirty clauses, and no operation ever adds new non-dirty clauses.
- There are no constraints on the ordering of phase one and two. If they would both eliminate a clause, any ordering would still eliminate it. Similarly, if they would both simplify a clause, any ordering would still simplify it. However, both phases should be placed before phase three because both may open opportunities for that phase. Finally, as phase four removes potential candidates for phase three, it should be performed at the end.

The time complexity of the algorithm is $O(n^3)$ due to $\mathcal{P}_3$ ($\mathcal{P}_1$ and $\mathcal{P}_2$ are $O(n^2)$ and $\mathcal{P}_4$ is $O(n)$). Here $n = max(\#I, \#U)$, where $\#I$ is the size of the invariant and $\#U$ is the size of the update. However, we cannot prove that the algorithm is guaranteed to *minimize* the set of variables. It is in fact an intractable problem to provide that guarantee, as shown in Sect. 5. At first we will establish the correctness of the algorithm, i.e., the guarantee that the simplified invariant evaluates to the same value as the original one.

4 Correctness

The correctness of all four phases, Thm. 1, has been mechanically verified in the proof assistant Coq [3] in about 8000 lines of Coq source code. For more details, please consult the project's home page `http://cs.au.dk/~gedefar/invariant`. We have referred to correctness as *soundness* as well, because it shows that the simplified invariant is satisfied in exactly the same cases as the original invariant, i.e., that the simplifications are sound. To simplify the notation slightly, we globally assume that the update U and data store D are wellformed, i.e., that $dom(U) \subseteq dom(D)$.

Theorem 1 (Phase Correctness). *Given I, U, D and assuming $wf(I, U, D)$ and $[\,], D \vdash I$, the following properties hold:*

$$U, D \vdash I \iff U, D \vdash \mathcal{P}_i[\![\, I \,]\!] \qquad \text{for } i \in \{1, 3, 4\}$$
$$U, D \vdash I \iff U, D \vdash \mathcal{P}_2[\![\, I \,]\!](U)$$

Corollary 1 (Correctness). *Given I, U, D and assuming $wf(I, U, D)$ and $[\,], D \vdash I$, the following holds:*

$$U, D \vdash I \iff U, D \vdash \mathcal{P}_4 \circ \mathcal{P}_3[\![\, \mathcal{P}_2[\![\, \mathcal{P}_1[\![\, I \,]\!] \,]\!](U) \,]\!]$$

The algorithm is in fact an optimization in that it reduces the number of variables to look up in the data store by reducing the number of literals (except in the worst case, where the invariant could not be simplified at all). Lemma 2 proves this optimization result.

Lemma 2 (Combined Reduction). *Given I, U, D, the following then holds*

$$|Vars(\mathcal{P}_4 \circ \mathcal{P}_3[\![\, \mathcal{P}_2[\![\, \mathcal{P}_1[\![\, I \,]\!] \,]\!](U) \,]\!])| \leq |Vars(I)|$$

Note that our case study furthermore supports the claim that the simplification provided by our algorithm is very good, in the sense that it is just as good as an optimization which is performed manually for each invariant by experienced engineers.

5 Complexity

In this section we characterize the complexity of the problem, showing that it is co-NP hard, but in $O(4^{\#V} \times \#I)$, where $\#V$ is the number of variables and $\#I$ is the size of the invariant. The ideal characterization of the problem that is being solved is as follows:

Definition 2 (Optimal invariant check). *Given a propositional expression I and a subset V' of the variables V in I. Find a minimal set of variables $V'' \subseteq V$ such that there exists a function i that maps each assignment of values to the variables in $V' \cup V''$ to a boolean value. The function i must be a perfect predictor for I, i.e., for an arbitrary assignment of values to variables in V it must return* `true` *iff I is satisfied.*

This means that it is correct to evaluate i on the values of $V' \cup V''$ as a substitute for evaluating I on the values of V. Moreover, this is an optimal solution under the assumption of expensive variable lookup, because $V' \cup V''$ is minimal and caching of old variable values is impossible.

We use the phrase *before the update* to describe an assignment that only differs on V', and note that I may or may not be satisfied before the update in this definition. It may seem likely that the problem would be easier if we require that there must exist a satisfying assignment before the update, i.e., that we only consider assignments for V where we can satisfy I by changing the values of some variables from V' only. As we shall see, though, both variants of the problem are co-NP hard.

Note that it would be uninteresting to consider the complexity of the problem when solutions are accepted that will definitely report violations, but might have false positives. In that case we do nothing (so the running time is constant), we choose $V'' = \emptyset$, and we always answer that the invariant might be violated.

The invariant check problem has an obvious solution, given enough resources. We use this to characterize the complexity of the problem by means of an upper bound.

Lemma 3 (Upper bound). *The problem of performing an optimal invariant check is in $O(4^{\#V} \times \#I)$ where $\#V$ is the number of variables V, and $\#I$ is the size of the invariant I.*

Proof. The terminology is as in Def. 2. We check each subset of V that includes V', ordered by size. For a given subset $V' \cup V''$, we consider all possible assignments of values to variables in V, and extend a truth table for $V' \cup V''$ with one extra column representing the function i. For each row, i.e., each choice of values in $V' \cup V''$, put `true` if I is `true` in all cases, i.e., for all choices of values in V, put `false` if I is `false` in all cases, and leave it blank if I is sometimes `true`, sometimes `false`. If the entire truth table has been filled in then $V' \cup V''$ is sufficient to precisely predict the value of I, but if there are any blank entries then this subset is insufficient and we continue with the next subset of V. This will produce a minimal subset $V' \cup V''$ of V that has a perfect predictor i of I, and it is easy to see that this algorithm evaluates I at most $2^{\#V-\#V'} \times 2^{\#V}$ times, where $\#V$ is the number of variables and $\#V'$ is the number of updated variables, such that the complexity is in $O(4^{\#V} \times \#I)$. □

This shows that a straightforward, naive algorithm exists, but the exponential complexity makes it impractical when V is anything but very small. However, it is not just this particular algorithm that has intractable running times.

Lemma 4 (Lower bound). *The optimal invariant check problem is co-NP hard; so is the variant of the problem where it is assumed that the invariant is satisfied before the update.*

Proof. Similarly to Cook [9], we use the problem of tautology to show the complexity of the invariant check problem. Let T be an arbitrary problem in TAUTOLOGY, i.e., a propositional expression for which we wish to decide whether it is true that all assignments will make T `true`. Choose a fresh variable p', and consider $T' = p' \vee T$. Let $V' = \{p'\}$ and find the minimal V'' such that $V' \cup V''$ has a perfect predictor t' for T'. If T is a tautology then $V'' = \emptyset$ and the truth table for t' will be `true` in its row for $p' \mapsto$ `false`. If T is not a tautology then the row for $p' \mapsto$ `false` will have the value `false`, or V'' will be non-empty. Note that we can trivially satisfy T' by setting p' to `true`, which means that this technique inherently satisfies the potential extra assumption that there must exist a satisfying assignment before the update. Hence, the optimal invariant check and even the seemingly weaker variant with the extra assumption will work as an oracle for deciding every problem in TAUTOLOGY. Since TAUTOLOGY is co-NP complete, the optimal invariant check problem and the variant are both co-NP hard. □

We conclude that the optimal invariant check problem—both with and without the extra assumption of satisfaction before the update—is hard. For comparison, another well-known and difficult problem is the integer factorization problem (the foundation of encryption), but as it lies in both NP and co-NP it is believed to not be as hard as the optimal invariant check problem (unless NP=co-NP).

The fact that the optimal invariant check problem as such is hard shows that we cannot expect to improve our algorithm to deliver the ideal, minimal solution (i.e., we cannot achieve completeness), and preserve a practical complexity. Yet, for very small problems it is of course always possible to use the algorithm described in the proof of Lem. 3.

6 Case Study

In this section we describe our approach to using the presented algorithm in a case study that we performed on the Google Maps backend. We first briefly describe invariants in Google Maps, then we describe how we represented the invariants in the approach, and finally we report on the reduction in object lookups achieved by using our algorithm.

The basic unit of data in Google Maps is traditionally called a *feature*. It represents an individual mutable entity with identity and properties, such as a road, a mountain, or a country. It is basically a simple object with fields, getters, and setters, but without inheritance and without methods containing general computational code. Our model does not need to express the setters as the invariants do not mutate the objects. These objects can be part of a *relationship*, i.e., they can hold references to each other in fields. Fields may also store primitive values such as integers. For example, if a mountain is located in a country, the object for the mountain may have a field which holds a reference to the object for the country, and another field which holds a primitive integer value indicating its height. Note that the updates are capable of building arbitrary object graphs as long as they respect the types of fields, which means that the complexity of the global object graph corresponds to the complexity of a heap in a general purpose object-oriented language.

There are invariants in place to make sure that objects are well-formed (e.g., that street addresses are formatted correctly) and that relationships between objects do not exhibit certain kinds of incorrectness (e.g., a mountain in a country must be located physically within the geographical border of that country).

In Google Maps the invariants are represented as a `C++` program, which is executed whenever the database is updated to reverify the invariants. An update to the database comes in the form of an updated object, including a specification of what information changed. When such an updated object is received by the database, the `C++` program verifies the updated object itself, and the relationships that the updated object are part of. Several updated objects can also be received together in a transactional style. Checking that the single object is well-formed is cheap, but checking a relationship is expensive, because it requires the program to fetch all the constituents of the relationship. Our algorithm was therefore introduced and used to minimize the number of object lookups when checking relationships.

The `C++` program is separated into a set of small functions that each represent an invariant taking an object and returning a boolean, signaling whether the object was valid or not. Furthermore each function trivially returns true if

the invariant checked by the function does not apply to the given type of object. In the mountain example the invariant only applies to objects of type mountain, and so the function implementing this invariant would trivially return true for non-mountain objects. These functions are combined, such that an object is valid iff all functions return true (i.e. it is a big conjunction). Each function requests objects that are in the relationship with the verified object, and the function is basically a list of comparisons between fields in the updated object and fields in the requested objects. The comparisons can involve arbitrary computation expressible in `C++`. For the mountain example, the function which verifies that a mountain is inside a country would request the country object for that mountain from the database during the execution, and in a single comparison it would verify that the location of the mountain is within the physical area of the country. The latter would typically be a library call. An update may affect several relationships and hence several of these functions.

Each of these functions represent an invariant in our approach and is the target for our optimization. For our approach, the important pieces of information in the functions are the comparisons and their location, as they represent the formal variables and collectively the formal boolean expression in our algorithm. To get this information, the author of the functions annotates the comparisons in it such that at runtime the framework can reconstruct the structure of the underlying boolean expression and the exact location of the comparisons in a test environment. The annotations are method calls to the framework which must be inserted into the C++ code. For instance, when comparing two values `a` and `b` for equality one must use `ops.areEqual(a, b)` rather than "`==`", where `ops` is an argument to the function implementing the invariant check. The invocation of `areEqual` will record that `a` and `b` are compared, and it will return the result of the equality comparison. Note that in the process of comparing two fields, arbitrary computation can be involved — as long as the annotations are in place to specify which fields are compared to which other fields. The comparisons that only use data from the remote database are considered non-dirty, whereas all other comparisons are considered dirty. Removing a clause in this implementation simply amounts to ignoring the comparison, because each comparison would output a debug message if the comparison failed.

In the process of reconstructing the underlying boolean expression, we use annotations and a dynamic test environment (using a test version of the `ops` argument mentioned above). This is because we need to track the individual objects and the comparisons in which they participate, in order to track the dirtyness of the comparisons. We should mention that the updated object corresponds to the update in our formal model, and the remote database corresponds to the data store in our formal model.

When validating an updated object our approach runs a three-stage process. In the first stage it executes each of the `C++` functions in a test environment on the updated object to construct the underlying boolean expression of the invariant. It then optimizes the expression and determines which requested objects to fetch from the database. In the second stage these requested objects are retrieved from

the database. Finally in the third stage the function is executed again, this time to verify the updated object, given the retrieved objects from stage two.

6.1 Experimental Setup

We evaluated the performance of our implementation by comparing the number of objects requested with implementations of two other algorithms: a *non-optimized*, and a *manually optimized* algorithm. The non-optimized algorithm performs a naive invariant check, and hence provides a base line for the comparison; it simply fetches all required objects and verifies the invariant of the relationships. The manually optimized algorithm provides a high quality solution, which was in use before our approach was introduced. It also uses a three stage process, but the first stage, compared to ours, is implemented by manually engineering the logic to determine whether fetching an object is necessary, given that the changed fields in the updated object are known. This logic is tailored to each invariant, and the logic may of course utilize arbitrary domain-specific knowledge and arbitrary algorithms that the engineer considers helpful. For instance, in the previously mentioned situation where it is verified that the mountain is inside the country boundary: if the country is enlarged by adding new areas to it there is no need to verify the relationship, since it is guaranteed to still be valid. The second stage and third stage of the manually optimized approach is similar to ours, by fetching and verifying the updated object using the requested objects from the first stage. Note though that the implementation for the third stage, in contrast with our approach, is different from the first stage.

These three approaches serve as the `C++` invariant program which is invoked when an update is received by the Google Maps database. As we were interested in the number of requested objects by these three approaches, we compared the number of requested objects in the first stage of our approach with that number in the non-optimized approach and in the first stage of the manually optimized approach.

We used a strict subset of the objects in the Google Maps database as the underlying data. Technically we selected 6 consecutive versions of the database and for each consecutive pair of versions we randomly selected 110 mil. objects. The object in the older version served as the pre-update object, and the newer version as the post-update object. From that we calculated the updated object which holds the post-update values along with indications of what fields were updated. The updated objects were used as input to each of the three approaches. In total we had 5 sets, of about 110 mil. updated objects each.

For each updated object in each of the five sets, we input the object into the three approaches, and counted the number of objects requested to verify the updated object. Finally we recorded the average size of the invariants.

6.2 Results

Table 1 shows the results of our experiment. Each row contains the result for a particular data set, as indicated by the first column, “Set”, where the last row

is the average of the preceeding rows. The second column, "Number of objects", contains the number of updated objects in the data set, as calculated from two consecutive versions of Google Maps. The third column, "Avg. inv. vars" denotes the average size of the invariants, measured as the number of variables in each invariant. Note that there were many different invariants involved, because many different types of objects had been updated. The next three columns, "Un-optimized", "Manually optimized", and "Our approach", contain the number of requested objects for the three algorithms. The seventh column, $\frac{\text{Unopt.}}{\text{Our}}$, shows the ratio of how many more lookups the un-optimized algorithm performs compared to our approach; and finally the last column, $\frac{\text{Man. opt.}}{\text{Our}}$, shows the ratio of how many lookups the manually optimized approach performs compared to our approach.

Table 1. Results from executing the three algorithms

Set	Number of objects	Avg. inv. vars	Objects needed to check invariant			Lookup ratio	
			Un-optimized	Manually optimized	Our approach	$\frac{\text{Unopt.}}{\text{Our}}$	$\frac{\text{Man. opt.}}{\text{Our}}$
1	110,719,642	309	245,049,813	3,604,078	3,934,770	62.3	0.916
2	110,154,090	306	246,689,891	4,651,570	4,910,443	50.2	0.947
3	110,982,143	237	264,520,677	12,742,286	12,943,598	20.4	0.984
4	109,995,628	308	260,988,659	21,362,611	21,444,239	12.2	0.996
5	110,987,702	276	264,567,342	12,724,787	12,922,880	20.5	0.984
Avg	110,567,841	287	256,363,276	11,017,066	11,231,186	33.1	0.966

For an easier comparison of the number of requested objects we have plotted the numbers from the three middle columns on a logarithmic scale in Fig. 3. The graph is categorized by the three algorithms.

At first one sees that the number of requested objects for the unoptimized approach is fairly stable throughout the data sets, ranging from 245 mil. to 264 mil. with an average of 256 mil. That is a difference of about 8%, whereas for both the manually optimized and our approach the numbers vary heavily, from about 3 mil. to 21 mil.; a factor of 6 in difference, with an average of 11 mil. Secondly we see that the ratio between the manually optimized and our approach are very close to 1, ranging from 0.996 in set 4 to 0.916 in set 1. The big difference in the number of requested objects for the manually optimized approach as well as our approach indicates that the amount of information changing from one data set to another data set varies, as the unoptimized approach requests about the same number of objects pr. data set. We have no reason to believe that the differences reflect a development over time, it just happened to be the case that the oldest changes enabled better optimizations than number 2–4, and the most recent changes were again somewhat more optimizable.

We note that our approach and the manually optimized approach indeed optimize the invariant check significantly, decreasing the number of requested objects

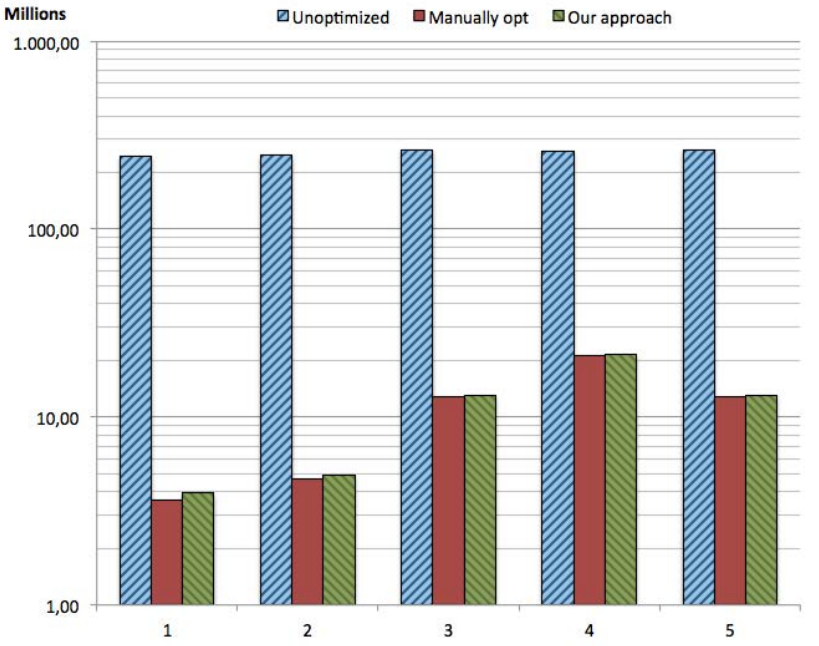


Fig. 3. The number of requested objects per approach. Note the logarithmic scale.

by an overall average factor of thirty compared to the unoptimized approach. Secondly the ratio between the manually optimized approach and our approach is very close to 1, namely on average 0.966. This is also clearly visualized in Fig. 3, where the two last pillars in each category are almost the same height. This small difference means that we get almost the same number of reductions as the manually optimized version, which indicates that the domain specific knowledge employed in the manual approach does not enable it to significantly outperform our approach.

Moreover, compared with the manually optimized invariant checks, our approach has the advantage that the person who specifies the invariant does not need to consider the logic for figuring out whether an object should be fetched or not. In contrast, the manually optimized invariant is specified in two "versions" of the check, one version for figuring out what objects to request, and one version to actually reverify the update based on the requested objects. The time overhead of our preprocessing step compared to the preprocessing step in the manual approach is about 55%, but the cost of preprocessing is negligible compared to the validation itself. In our current implementation, we need a few annotations to control this three-stage approach, but in future work we hope to limit the burden such that the annotations can be avoided. Because of the good performance, the technique is now aimed for production for validating incoming updates to Google Maps.

7 Discussion

In this section we will discuss a few additional issues, including different approaches to fetching the values of variables when a simplified invariant has been obtained.

First, we should note that some trivial simplifications can be made along with the transformation of the query into CNF. They are not required for correctness, but they may reduce the size of the invariant, or they may reveal that it is unsatisfiable or always satisfied. In particular, no disjunctions should contain both x and $\neg x$ for any x; in this case the corresponding clause is trivially `true` and may be omitted, and if all clauses are thus omitted the invariant is always satisfied. There should not be any clauses which are exact negations of each other; in this case exactly one of them is `false` for every possible assignment, and the invariant is unsatisfiable.

It may be possible to infer more about the values of individual variables than what we do in the current algorithm. For instance, we could exploit the existence of both $x_1 \vee x_2$ and $\neg x_1 \vee x_2$ in the original invariant to get the value of x_2 in the data store: No matter whether x_1 is `true` or `false`, we can only satisfy both of these clauses if x_2 is `true`. In general, going down this path means creating and proving theorems about the given variables, and it is not obvious whether it will be more effective than running the general algorithm from the proof of Lem. 3. We have chosen to avoid using such proof based elements, but it may be useful to heuristically select and use a set of simple theorems, which would then give rise to an additional phase in the algorithm, and possibly a new ordering. Section 8 briefly dissusses this issue, too.

We have not yet discussed how to use the output from our algorithm, but this is actually not trivial, either. When the algorithm has been executed, the literals in the simplified invariant contain the set of variables whose values we need to look up in the data store in order to reverify the invariant. It may be important for the performance how we fetch the values of these variables.

One approach would be to simply fetch the variables one at a time. This would be appropriate if the data store is in shared memory, or the main point is that we have eliminated the need to fetch some variables that are not accessible at all, and the rest of the variables are easy to get at. On the other hand it might be more appropriate to fetch all the variables in one operation if the data store is a database on a remote server, and the main cost of fetching variables is the creation and destruction of a database connection rather than the actual data transfer. In the following we discuss both scenarios.

If the situation requires that the variables are fetched one at a time, the lookup order of the variables is important, because one may reduce the need for further lookups by using the (now known) value of each newly fetched variable to run the algorithm again. Given that such a variable has a known value which is furthermore known to be the value from before the update, we may obtain new non-dirty unit clauses and also simplify clauses in phase two, and the entire algorithm may thus obtain significantly better results. We do not know whether it is better to start the algorithm from scratch or continuing with the already

simplified invariant, but the latter would require some extra bookkeeping in order to recognize the additional non-dirty unit clauses correctly. Due to the inherent complexity of the problem we do not believe that it is possible to create an optimal plan for which variable to fetch first, so that decision will have to be made based on various heuristics.

One option is to aim for reducing clauses to $\top$, by looking for literals that occur in many clauses. The variables of these literals could be prioritized over the other variables, such that if a literal is evaluated to `true` these clauses are instantaneously reduced to $\top$. If instead the literals are evaluated to `false` then at least all of the evaluated literals can be removed from their containing clauses. On the other hand one could aim at making one clause simplify to the empty clause (i.e. the never satisfied clause), by searching for small clauses that do not require many lookups to be invalidated. As soon as an empty clause arises, the invariant is guaranteed to be violated and the algorithm can terminate. Finally, one could aim at establishing subset relations by looking up variables occurring in small clauses where the removal of this variable would create a subset relation to one or more other clauses, which would then be eliminated by a run of phase three. It may, however, be a non-trivial task to discover this type of situation.

One useful factor to consider when choosing a lookup order is the probability of individual values. If, say, a `true` value is much more common than a `false` value (globally or for a specific variable) then we could give more weight to the outcome of having `true` than the outcome of having `false`. Such information could be used to prioritize positive variables (i.e., variables that occur in positive literals) over negative variables, because we would be more likely to satisfy a clause based on the former than the latter. Similarly, with a focus on empty clauses we could prioritize negative variables because they would have a high probability of being `false` and hence making small clauses even smaller.

In situations the cost of fetching n variables and the cost of fetching $n + 1$ variables are almost the same, we should consider fetching all the required variables in one step. Even when fetching a subset of the required variables could enable further simplifications, it might be more costly to fetch the variables in two batches, no matter how helpful the first batch turns out to be.

In cases where the extra cost of fetching several variables is negligible compared to fetching one variable, it might still be much more expensive to fetch many variables, due to bandwidth, congestion, etc. A hybrid approach, where a subset of the variables are fetched together is then desirable. Using this scheme, one could pick all the variables in one clause at a time to look for clauses to invalidate. Or fetch variables that span all clauses and thereby either satisfy or reduce all clauses.

Obviously, there are many ways to do it, and the many trade-offs may be good or bad depending on a large number of complex and dynamic properties of the environment and data values. This means that there is ample room for heuristics and manual optimizations, and this again fits well with the knowledge that an ideal solution to the underlying problem is intractable.

One issue we haven't discussed here is the situation where the cost of fetching a variable varies. For an in-depth treatment of this situation, though in the context of SAT solvers, we refer to Klaus Truemper [24].

8 Related Work

Optimization of invariant checking is an important area within the database community. Reducing the number of lookups means that it is less costly to check all the invariants and reevaluate the derived views of any of tables in the database. The problem of detecting the effects of an update has therefore been studied intensively in the database community. Some researchers have focused on investigating when an update is irrelevant with respect to, or independent of, an invariant [5,15,16,17,22]. This is also called incremental integrity checking or constraint simplification [18]. Others have focused on whether an update is irrelevant with respect to a derived view of a table in the database [4,12,23]. Generally speaking, verifying the irrelevance or independence of a query with the same level of expressive power as Datalog is undecidable. In many subcases [4,12,21], including cases where the query language is less expressive than Datalog, it is however decidable. In the following we will discuss some of the many techniques that are most closely related to our approach. For more details, and a good overview, we refer to Bry et al. [7] or Gupta et al. [14].

There is an early paper by Hammer and Sarin [16] mentioning that they use logical analysis of database assertions for certain types of updates to construct assertion violation checks that can be evaluated efficiently. The paper is only an abstract giving no details at all, and there are no follow-up papers by any of these authors that offer more insight into the details of their technique.

Nicolas [22] proposes a method for optimizing database invariants given as predicate logic expressions. The method basically performs phase 2 and phase 4 of our algorithm by inserting the values from the updated database row into the invariant, assuming that the invariant was satisfied before the update. This paper deals with predicate logic, but there are some cases, such as existentially quantified variables, that are not treated. Finally, the formalization is not associated with mechanically verified proofs.

The work by Blaustein [5] is similar to Nicolas [22], except that he also treats existentially quantified variables. However it is shown by Hsu & Imielinski [17] that some of the methods by Blaustein are not correct in all situations.

Blakeley et al. [4,23] proposes techniques for detecting when an update to a table in a database is irrelevant for a view of the table. If the update is relevant for the view, they compute whether the update only requires access to information stored in the view. If so, the update is also committed to the view, without the need for recomputing the view and thereby accessing the underlying table. They consider the problem both at compile-time and at run-time. Besides having an exponential complexity, their approach does not try to minimize the information needed from the table, if the view must be recomputed and some information from the table is required. Our algorithm is polynomial and minimizes the information needed from the table, if any.

Charles Elkan [12] studies when an update is irrelevant wrt. to a query (e.g. view or invariant), but in comparison with Blakeley et al. [4] does so in a formal setting, also considering recursive queries. Compared to our approach, his work is not mechanically verified and he does not treat the simplification part of an query if it is not independent.

Gupta & Widom [15] study invariant checking in a distributed database setting, where a global invariant is in place. Using information from an update and a local database, they show how to reverify the invariant using data from the local database only, if possible. If it is not possible, their technique reverts back to a normal invariant check. In comparison to our approach, their approach has access to information in a local database, and their approach reverts to a full invariant check if the local data is insufficient. Our algorithm on the other hand reduces the information fetched from the remote data store.

Another area that has certain similarities to our algorithm is partial evaluation, e.g. [19]. Partial evaluation is a technique for optimizing programs by exploiting that certain values computed by a program are known at compile time, also known as the static parts. The parts not known at compile time are known as the dynamic parts. Constant folding, where the values of constant expressions are computed at compile time, is a simple instance of partial evaluation. For our algorithm one could think of the updated variables as the static part, whereas the variables whose values are in the data store constitute the dynamic part. Typically, partial evaluation works with Turing-complete languages where termination is a big issue that must be delicately dealt with. Our algorithm, on the contrary, works with propositional logic where termination is easily achieved, and where we may exploit additional properties associated with this particular type of data, such as the subset property of phase three. We believe that this exploitation of type-specific knowledge goes beyond what is used in the context of partial evaluation.

Even though there are some similarities between our algorithm and SAT solvers [13], the underlying problem is completely different. Our algorithm assumes that the boolean formula had a satisfying assignment before the update, and now the problem is to reduce the set of variables we have to retrieve. In SAT solving, one does not know whether a formula has a satisfying assignment, and the job of the SAT solver is find such an assignment if it exists, and otherwise report that no such assignment exists. The complexity of the two problems also signifies the difference, as minimizing is co-NP hard, whereas SAT solving is NP-complete. On the other hand, the commonalities among these two topics are so significant that they should be considered, and so we have adopted the notation from SAT solving and used techniques known from SAT solving, such as utilizing unit clauses [26].

Reoptimization [6] is the problem of finding an optimal solution for a problem instance P' which is obtained from a problem instance P by a suitably defined 'small' modification, starting from a given, optimal solution to P. A similar issue is the problem of finding the minimum number of gates to reevaluate in a boolean circuit, given a set of updated input bits [10,25]. Both problems are similar to our

algorithm in that we handle modifications to problems, but they are still very different at the core because we do not rely on knowing the optimal solution to the original version of the problem, respectively knowing the pre-update input.

A number of approaches enable some compile-time analysis of properties of program execution. For instance, Spec# [2] is a formal specification language for C# that enables annotating code with pre- and postconditions, invariants, and other assertions. Using Boogie [1] and the SMT solver Z3 [11], it is possible to prove or refute some of these annotations at compile-time. This may involve the automatic generation of proofs about the value of Boolean expressions, which creates an obvious connection to our work. A similar set of tools and approaches exists for Java, such as JML [20] and ESC/Java2 [8]. In general, we do not think that these approaches will directly support solutions to the invariant check problem, because they focus on proving given theorems rather than finding equivalent but simplified ones. But it would certainly be possible to use them to check certain potential solutions. In particular, if E is a simplified invariant produced by our algorithm then $\neg(I \Leftrightarrow E)$ would be unsatisfiable iff E is a perfect predictor of I. The fact that the invariant has already been significantly simplified helps keeping the cost of this operation low, and the solver adds in other techniques than the ones that we apply, so a hybrid approach could be very promising.

9 Conclusion

We have presented an analysis of the potential simplifications that may be achieved when an invariant is known to hold for a set of variables, and an update is applied to some of these variables, and we wish to check whether the invariant is violated by this update. The starting point is the assumption that it is expensive to fetch the value of a non-updated variable, and hence we wish to check the invariant based on as few variables as possible. Our approach supports the constraint that it is impossible to cache the values of all variables in the invariant, which is required in order to make the technique applicable with objects, and invariants that apply to specific types of objects, as in our case study. Our main contribution is an algorithm in four phases which is proven correct by a mechanically checked proof in Coq. The algorithm is sound, i.e., it will produce a simplified invariant with the same truth value as the original invariant, hence making it possible to check for invariant violations based on a smaller set of variables. The algorithm is not complete, i.e., it may produce an invariant whose set of variables is not minimal. However, we prove that the underlying problem is co-NP hard , and hence we cannot hope for a complete solution provided by an algorithm with an acceptable running time. We do provide a fully general algorithm, though, that may be used with very small sets of variables. We report on a case study for Google Maps, which describes a solution using the presented technique. This solution is now aimed for production at Google. It shows that our algorithm is indeed useful, yielding an improvement of a factor of thirty in the number of requested objects compared with an unoptimized implementation, and providing almost identical improvement (within 3.5%) compared with the

ones obtained by a solution which is optimized manually. Our algorithm thus eliminates the need for the painstaking manual optimization work.

Acknowledgments. We would like to thank Peter A. Jonsson, Holger Flier and Marc Nunkesser for comments on earlier versions of this paper.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, vol. XXV. Springer (2004)
4. Blakeley, J.A., Coburn, N., Larson, P.-Å.: Updating derived relations: Detecting irrelevant and autonomously computable updates. ACM Trans. Database Syst. 14(3), 369–400 (1989)
5. Blaustein, B.T.: Enforcing database assertions: Techniques and applications. Number 21-81. Thesis, Ph.D (1981)
6. Böckenhauer, H.-J., Hromkovič, J., Mömke, T., Widmayer, P.: On the hardness of reoptimization. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 50–65. Springer, Heidelberg (2008)
7. Bry, F., Manthey, R., Martens, B.: Integrity verification in knowledge bases. In: Voronkov, A. (ed.) RCLP 1990 and RCLP 1991. LNCS, vol. 592, pp. 114–139. Springer, Heidelberg (1992)
8. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
9. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC 1971, pp. 151–158. ACM (1971)
10. Cordy, J.R., Halpern-Hamu, C.D., Promislow, E.: TXL: a rapid prototyping system for programming language dialects. Comput. Lang. 16(1), 97–107 (1991)
11. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. Elkan, C.: Independence of logic database queries and updates. In: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville, Tennessee, April 2-4, pp. 154–160. ACM Press (1990)
13. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Chapter 2 satisfiability solvers. In: van Harmelen, V.L.F., Porter, B. (eds.) Handbook of Knowledge Representation. Foundations of Artificial Intelligence, vol. 3, pp. 89–134. Elsevier (2008)

14. Gupta, A., Sagiv, Y., Ullman, J.D., Widom, J.: Constraint checking with partial information. In: Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Minneapolis, Minnesota, May 24-26, pp. 45–55. ACM Press (1994)
15. Gupta, A., Widom, J.: Local verification of global integrity constraints in distributed databases. In: SIGMOD Conference, pp. 49–58. ACM Press (1993)
16. Hammer, M., Sarin, S.K.: Efficient monitoring of database assertions (abstract). In: Lowenthal, E.I., Dale, N.B. (eds.) Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 31-June 2, p. 159. ACM (1978)
17. Hsu, A., Imielinski, T.: Integrity checking for multiple updates. In: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, SIGMOD 1985, pp. 152–168. ACM (1985)
18. Ibrahim, H.: Checking integrity constraints - how it differs in centralized, distributed and parallel databases. In: 17th International Workshop on Database and Expert Systems Applications, DEXA 2006, pp. 563–568 (2006)
19. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall international series in computer science. Prentice Hall (1993)
20. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Kiniry, J.: JML reference manual (June 30, 2004)
21. Levy, A.Y., Sagiv, Y.: Queries independent of updates. In: Proceedings of the 19th International Conference on Very Large Data Bases, Dublin, Ireland, August 24-27, pp. 171–181 (1993)
22. Nicolas, J.-M.: Logic for improving integrity checking in relational data bases. Acta Inf. 18, 227–253 (1982)
23. Tompa, F.W., Blakeley, J.A.: Maintaining materialized views without accessing base data. Information Systems 13(4), 393–406 (1988)
24. Truemper, K.: Design of Logic-based Intelligent Systems. Wiley-Interscience publication, John Wiley & Sons (2004)
25. Vyatkin, V.: Event-driven traversal of logic circuits for re-evaluation of boolean functions in reactive systems. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 319–328. Springer, Heidelberg (2004)
26. Wos, L., Carson, D., Robinson, G.: The unit preference strategy in theorem proving. In: Proceedings of the Fall Joint Computer Conference, Part I, AFIPS 1964 (Fall, Part I), October 27-29, pp. 615–621. ACM (1964)

Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions

Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers

ETH Zurich, Switzerland
stheule@ethz.ch,
{ioannis.kassios,peter.mueller,alexander.summers}@inf.ethz.ch

Abstract. Abstract predicates are the primary abstraction mechanism for program logics based on access permissions, such as separation logic and implicit dynamic frames. In addition to abstract predicates, it is useful to also support classical abstraction functions, for instance, to encode side-effect-free methods of the program and use them in specifications. However, combining abstract predicates and abstraction functions in a verification condition generator leads to subtle interactions, which complicate reasoning about heap modifications. Such complications may compromise soundness or cause divergence of the prover in the context of automated verification. In this paper, we present an encoding of abstract predicates and abstraction functions in the verification condition generator Boogie. Our encoding is sound and handles recursion in a way that is suitable for automatic verification using SMT solvers. It is implemented in the automatic verifier Chalice.

1 Introduction

Program logics based on access permissions, such as separation logic [24] and implicit dynamic frames [27] are the foundation of many program verifiers for heap-manipulating programs [5,11,15,21,26]. They associate an access permission with each heap location, and enforce that a method accesses a location only if it has the permission to do so. To enable modular verification, each method specification states which permissions the method requires from its caller (in its precondition) and returns to its caller (in its postcondition). Upon a call, the caller relinquishes the required permissions (we say the caller *exhales* the precondition) and transfers them to the callee (the callee *inhales* them). Conversely, when a method terminates, the method exhales its postcondition, while its caller inhales it. This technique simplifies framing; as long as a method holds on to the permission for a location (that is, does not exhale it), no other method can access that location and, thus, its value remains unchanged. When permission to a location is lost, the value recorded for that heap location should also be discarded; in the common parlance, this (outdated) information must be *havoced*.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 451–476, 2013.

```
class List {
  var value: int;
  var next: List;

  predicate valid { acc(value) && acc(next) && (next ≠ null ⇒ next.valid) }

  function length(): int
    requires valid;
    ensures result > 0;
  { unfolding valid in next = null ? 1 : 1 + next.length() }

  function itemAt(i: int): int
    requires valid && 0 ≤ i && i<length();
  { unfolding valid in i = 0 || next = null ? value : next.itemAt(i-1) }

  method set(i: int, v: int)
    requires valid && 0 ≤ i && i < length();
    ensures valid && length() = old(length()) && itemAt(i) = v;
    ensures ∀ j in [0..length()-1] • i ≠ j ⇒ itemAt(j) = old(itemAt(j));
  {
    unfold valid;
    if (i = 0) { value := v; }
    else       { call next.set(i-1, v); }
    fold valid;
  }
}
```

Fig. 1. A Chalice [21] implementation of a singly-linked list. Methods have preconditions (keyword **requires**) and postconditions (keyword **ensures**). In addition to regular methods, Chalice supports side-effect-free functions, which may be used in specifications. An access permission to a field $o.f$ is denoted by **acc**($o.f$), which corresponds to $o.f \mapsto \_$ in separation logic. The Chalice conjunction && treats permissions multiplicatively (i.e., requiring the sum of the permissions in each conjunct), similarly to the separating conjunction $*$ of separation logic. The recursive abstract predicate `valid` represents the memory locations of the list structure. The **unfold** and **fold** ghost statements replace a predicate by its body and vice versa. The ghost expression construct **unfolding**...**in** is intuitively analogous to an **unfold**-**fold** block and can be used in functions and in specifications, where statements cannot occur.

Abstract Predicates. Enumerating all locations for which a method requires or returns permissions is not possible for recursive data structures. For instance, a method that traverses a linked list, such as method `itemAt` in Fig. 1 would require permission to access `this.value`, `this.next`, `this.next.value`, and so on. To solve this problem, Parkinson and Bierman [23] introduced abstract predicates. The definition of an abstract predicate declares a predicate body that may contain permissions to concrete heap locations, constraints on their values, and possibly further predicate instances. Due to this recursion, abstract predicates potentially represent permission to an unbounded number of heap locations. For instance, the abstract predicate `valid` in Fig. 1 represents the permissions for `value`, `next` and, if `next` is non-null, the permissions in `next.valid`. Just as with permissions to field locations, a method may require predicate instances from its caller. It may access a location if it possesses the corresponding permission,

either directly or as part of a predicate instance. For example, method `itemAt` requires the predicate instance `valid` to get access to all locations of the list.

In this paper, we employ ghost locations to describe the predicates held for a particular object (cf. Sec. 3). Holding an instance of a predicate is represented by holding permission to its ghost location. We use the term *location* for both concrete field locations and predicate locations. We use the term *permission* to describe permissions to both kinds of locations.

Abstraction Functions. Abstraction functions [14] map the concrete representation of a data structure (say, a linked list) to an abstract value (say, a sequence). Specifications can then be expressed in terms of the abstract values, which is important for information hiding. In the form of side-effect-free inspector methods (pure methods), abstraction functions are a key ingredient of contract languages such as Eiffel, JML, Spec#, and .NET CodeContracts, which support runtime assertion checking in addition to static verification.

Many permission logics express data abstraction via parameterised abstract predicates whose parameters represent abstracted values of a data structure and whose bodies relate the concrete representation of the data structure to these values. However, there are several advantages to supporting classical abstraction functions *along with* abstract predicates: (1) Most data structure implementations include side-effect-free inspector methods (or *functions* in Chalice) such as `itemAt` and `length` in our example. It is convenient to re-use these methods in specifications; side-effect-free methods can be encoded naturally as abstraction functions [8]. (2) Declaring abstraction functions does not affect the signatures and definitions of abstract predicates. This allows additional abstractions to be added to a library during maintenance, without changes to the predicates and re-verification of existing client code. (3) Specifications written without abstraction functions typically use logical variables for the parameters of an abstract predicate, which can then be used in postconditions to describe how the abstract value has changed. Finding witnesses for these logical variables is not supported well by SMT solvers[1]. By contrast, abstraction functions can be used within **`old`** expressions to refer to their pre-state evaluation, avoiding logical variables.

Abstraction function definitions must somehow be made available to the prover, so that it can relate a function application to the function's body. Since abstraction functions can be recursive, it is not possible to statically inline such definitions. In the recursive case one must also prevent the prover from unrolling the function definition infinitely often. A commonly-used approach is to use uninterpreted functions along with an axiom that relates the function to its body. In this case, the prover might select and apply such axioms infinitely often, which is known as a *matching loop* [9]. In our running example, unrolling the definition of the pure function `x.length()` would yield the expression `x.next=`**`null`**` ? 1 : 1+x.next.length()`, in which another call to `length` occurs.

[1] Existing tools that support rich parameterised predicates use custom reasoning engines based on symbolic execution, rather than verification condition generation.

Verification Condition Generation Verification Condition Generation (hereafter, VCG) is a popular technique for the construction of automated verifiers [4,7,12,17,18,20], in which the problem of verifying a program is encoded as a logical formula, and then handled by automatic theorem provers (typically SMT solvers). Since SMT solvers reason at a purely logical level, many problem aspects such as the program's heap state, and (for permission logics) auxiliary state to track the permissions currently held by a thread, are encoded as mathematical maps (total functions). Despite this (total) representation, a verifier can still take care that values of heap locations are only directly referred to when appropriate permissions are held. For concreteness, we present our approach in the context of Chalice, a VCG-based verifier for concurrent, imperative programs. Chalice provides only non-parameterised predicates, along with (parameterised) abstraction functions. In this setting, the primary role of predicates such as `valid` in Fig. 1 is to abstract over access permissions, whereas functions such as `length` and `itemAt` abstract over the contents of a data structure. Our results apply also to frameworks with parameterised predicates.

Contributions. The main contribution of this paper is an encoding of abstract predicates and abstraction functions for verification condition generators that is sound, and amenable to automation via SMT solvers.

To achieve these goals, we need solutions to the following encoding issues: (1) how to define which permissions are part of a recursive predicate instance (for example, to determine which permissions are transferred when it is exhaled), (2) how to define the values of recursive abstraction functions, and (3) how to define which heap locations the value of a recursive abstraction function depends on (for example, to determine whether a heap update affects the value of a function application).

The key insight motivating our solution is that, although each of the three issues above has, in principle, to do with a statically unbounded number of unrollings of a recursive definition, the unrolling of these definitions is typically only relevant up to the depth at which the program to be verified (including its contracts) has explicitly inspected the corresponding data structure, at either the current or an earlier program point. That is, our solution focuses on ensuring that recursively-defined information is made available for predicate and function bodies which have been *syntactically observed* at some program point up to the current one. Our solution employs some existing ideas, but combines and enhances them in an original way to achieve the first verification condition generator that avoids matching loops and is sound[2] (the previous version of the Chalice tool was unsound, due to an incorrect encoding of abstract predicates and abstraction functions). In particular, we present a novel encoding of those permissions inside a predicate that are needed to express proof obligations.

[2] Some verifiers based on symbolic execution [26,16] support both abstract predicates and abstraction functions, and many more support only the former [11,15,5]. In the equally important domain of verification condition generation, there is no solution that supports both features, is sound, and avoids matching loops.

We present our solution for implicit dynamic frames [27], but it also applies to other permission logics, such as separation logic. We have implemented our solution in a new version of Chalice.

Outline. Secs. 2 and 3 present our solution informally. The details of our encoding are explained in Secs. 4 and 5, and we argue why our solution is sound in Sec. 6. We discuss related work in Sec. 7 and conclude in Sec. 8.

2 Abstract Predicates

In this section, we describe informally how our technique deals with the first issue described in the contributions.

2.1 Folding and Unfolding

Whenever a method attempts to access a heap location, the verifier needs to check whether the method has the access permission for that heap location, either directly or as part of a predicate. However, since the definitions of predicates may be recursive, a verifier cannot determine precisely which permissions are part of a predicate; since the recursion is (statically) unbounded, it is neither possible to inline the predicate's body fully, nor is it useful to let an SMT solver reason directly about recursive definitions in an unconstrained manner.

Many verifiers [15,21,26] work around this problem by distinguishing between a predicate and its body. Instead of letting the SMT solver expand predicate definitions automatically, the verifier expands only specific predicate definitions at specific points in the program execution. *Unfolding* replaces a predicate by its body, while *folding* has the inverse effect. Until a predicate has been unfolded, the permissions and information implied by the predicate's body are generally not made available to the prover. We call a permission that has not been folded into a predicate instance *direct*; all other permissions are called *folded*. Accessing a field, unfolding a predicate, or exhaling a predicate all require appropriate direct permissions.

The method `set` in Fig. 1 illustrates these concepts. The method body first inhales its precondition; in particular, the predicate `valid`. After the inhale, it holds `valid` as a direct predicate, whereas the permissions to `value` and `next` (as well as the fields of the rest of the list) are folded. The **`fold`** statement at the end of the method is necessary to regain direct permission to `valid`, which gets exhaled as part of the postcondition.

Folding and unfolding transforms the problem of deciding how deeply to unroll a recursive definition to the problem of how deeply to unfold a predicate instance. Some tools provide heuristics for inferring unfold and fold operations, while others require programmers to indicate these operations through ghost statements. For our approach, it is irrelevant whether the unfold and fold operations are indicated explicitly or inferred, so long as there are specific points in

the program execution at which the transition between a predicate and its body takes place. In our examples, we use explicit ghost statements for clarity.

It may be tempting to think that **`fold`**/**`unfold`** statements alone solve our encoding problem. Indeed these statements, explicit or inferred, protect the SMT solver from the recursion in the predicate definitions. However, a challenge that remains is *when and how havocing happens*. In particular, it is not useful to havoc locations immediately when the *direct* permission to the location is released during a fold because abstraction functions provide a way of inspecting memory locations whose permissions have been folded. If we were to havoc locations upon **`fold`**, then a function that inspects a location after is has been folded into a predicate would return an arbitrary value, which would defeat the purpose of abstraction functions. For instance, the call to `itemAt` in the postcondition of `set` would yield an unknown value, and the postcondition could not be verified.

Not havocing locations when their permissions get folded into a predicate complicates the exhale operation: when a predicate gets exhaled, we have to havoc every location that is folded under it because permission to these locations is no longer held by the current method, neither directly nor folded. For example, when exhaling the predicate location `this.valid` (Fig. 1), we must havoc `this.valid` and every location of the linked list folded under it. This set of locations is not statically known and we must find a sound way to approximate it.

Our approach to the implementation of exhale consists of two stages. First, we havoc *very aggressively*: each time a predicate is exhaled, we havoc *all* memory locations for which the method does not retain a direct permission. This ensures soundness, but havocs too much: in fact it havocs all locations whose permissions are folded. Second, we make this crude approximation more precise, by recovering an underapproximation of the location values to which the method has folded permission after the exhale.

The key insight mentioned in the introduction can now be made more concrete: although some information about the contents of recursively-defined predicates and functions is essential, we need concern ourselves only with those predicate instances whose bodies were unfolded earlier in the program text. Therefore, we can focus on recording detailed information for precisely these instances.

2.2 Framing of Locations—Known-Folded Permissions

When the verifier has knowledge of the value of a heap location earlier in the program, it is important to preserve this information even though the permission to this location is now folded inside a predicate instance. This is a particular case of our earlier insight: it is not *all* folded permissions that we care about, but those (a) for which the program previously held the direct permission and (b) which are folded inside a predicate instance that has been retained up to the current program point. We call these permissions the *known-folded permissions* of a predicate instance. In our encoding, in addition to recording direct permissions, we record for each predicate instance those locations to which the predicate holds known-folded permission.

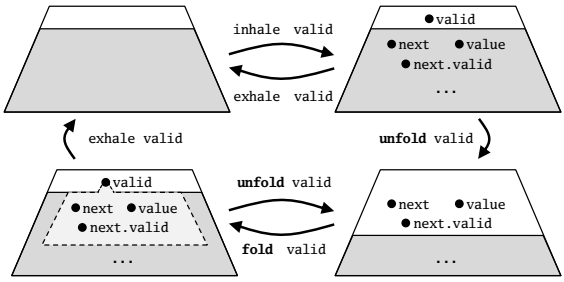


Fig. 2. Each trapezoid depicts all permissions held at a particular point in a method invocation, where the white regions contain direct permissions and the gray areas depict folded permissions. Known-folded permissions appear in the light gray region with dashed border (connected to the predicate they belong to).

Known-folded permissions provide an under-approximation of folded permissions, which (unlike the latter) can be precisely tracked in our encoding. A predicate instance contains known-folded permission to a location exactly when the direct permission to the location was folded (possibly deeply) inside the predicate instance at some earlier program point, either through a **fold** statement or through an **unfolding** expression. Because the known-folded permissions are always a subset of the folded permissions held by the current method, when exhaling a predicate it is sound to havoc only those locations for which the method holds neither direct nor known-folded permission.

Fig. 2 illustrates how our operations modify some of the possible states of the `valid` predicate. The upper row depicts states in which the values of fields `value` and `next` have not been observed; in the upper right trapezoid, the permissions in the body of `valid` are folded, but not known-folded. In the lower row the values of these fields have been observed, and permission to them is either direct or known-folded, which protects the fields from being havoced by an **exhale**.

As an example, consider two distinct `List` objects `x` and `y`, for which the `valid` predicate is held at the beginning of the execution of the following code:

```
var i: int := unfolding x.valid in x.value;
var j: int := unfolding y.valid in y.value;
call y.set(0,10);
assert unfolding x.valid in (i = x.value); // succeeds
assert unfolding y.valid in (j = y.value); // correctly fails to verify
```

The **unfolding** expressions at the beginning of the code make the permissions to `x.value` and `y.value` known-folded. The call to `y.set` exhales `y.valid` and thereby removes the permission to `y.value` from the known-folded permissions, since `y.value` is folded inside `y.valid`. It havocs the state, but preserves locations for which the method holds known-folded permission, in particular, `x.value`. This makes the first assertion succeed, while the second correctly fails to verify.

If we did not havoc aggressively, the second assertion of the example would verify, which is unsound. On the other hand, without keeping track of known-folded permissions, the first assertion would fail to verify, which is too imprecise.

3 Abstraction Functions

In this section, we explain how we handle the other two issues mentioned in the introduction, i.e., the definition and framing of abstraction functions.

3.1 Definition of Abstraction Functions

We encode each abstraction function as an uninterpreted function symbol together with a *definitional axiom* that relates the function symbol to the corresponding body. For recursive functions, this axiom might lead to a matching loop if the instantiation of the axiom is not controlled. Our approach lets the prover unroll the function definition only a statically known number of times.

A typical recursive abstraction function such as `length` in Fig. 1 requires permission to the locations of the data structure in its precondition (often via a predicate) and recurses on the data structure to compute its result. In particular, the body of such a function typically unfolds a predicate from its precondition before recursing on the next node in the structure via the same function (cf. `length`). The definitional axiom of such a function relates a function application that depends on a predicate to an expression that depends on the locations whose permissions are folded inside the predicate's body. Therefore, an instantiation of the axiom will typically only provide useful information if the verifier has observed some information about these contents. For instance, if a method does not have direct or known-folded permission to `o.next` and `o.next.valid` then it cannot observe the values of `o.next` and `o.next.length()`. Without any knowledge about these values, the definitional axiom for `o.length()` is not useful.

This observation leads us to tie the instantiation of function definitions to the occurrence of predicate instances which have been unfolded at some earlier point in the program code. Essentially, we again follow our key insight; it is enough to ensure that functions are unrolled to at least the depth that the corresponding predicate instances have been unfolded (at some program point up to the current one). Thus, the instantiation of axioms proceeds in lockstep with the method's traversal of the corresponding data structure.

The following example illustrates our approach. Assume that `unroll` is an additional method of class `List`. The method precondition mentions `length`, which allows the prover to expand the recursive definition only one level deep to avoid a matching loop. Therefore, the prover cannot conclude from the precondition that `next.next` is **`null`**. However, since the method cannot observe the value of `next.next` at this point (the permission is folded inside `valid`), the information would be useless anyway. The second **`unfold`** statement unfolds the predicate instance required by the function application `next.length()` and, therefore, allows the prover to instantiate the definitional axiom for this application, which provides the information necessary to verify the assertion.

```
method unroll()
  requires valid && length() = 2
{
  unfold valid;
  unfold next.valid;
  assert next.next = null;
}
```

The approach outlined above avoids matching loops by controlling the instantiations of recursive definitional axioms. It supports the typical uses of abstraction functions, but does not handle recursion that is not tied to the traversal of a data structure (e.g., a factorial function). The automatic handling of such functions is beyond the scope of this paper.

3.2 Termination of Abstraction Functions

A common problem in allowing arbitrary recursive definitions in specifications, is the potential for introducing unsoundness to the logic, via non-well-founded recursion. For example, the definition of a function $f() = 1 + f()$ is inconsistent and therefore must be forbidden.

The usual approach to handling this problem is to insist on the existence of a well-founded *termination measure* for each recursive function definition provided. In our approach, a function passes the termination-check if the verifier can show that every recursive function call is made within the body of an **unfolding** expression (that is, after unfolding a predicate instance). The number of predicate instances held defines a well-founded measure because every predicate instance can be unfolded only finitely many times during the execution of a program. States at which an infinite predicate instance is held are unreachable[3]. Notice that the termination measure described above permits the definitions of the abstraction functions in our running example.

It is important to note that this treatment of predicates does not rule out cyclic *heap structures*. For instance, doubly-linked structures can be easily handled with fractional permissions, while cyclic lists can be handled (as in separation logic) using list *segment* predicates[4].

3.3 Framing of Abstraction Functions

Since an SMT solver cannot unroll a recursive function definition arbitrarily deeply, it cannot use the function definitions to *frame* function applications, that is, to determine whether a heap update potentially affects the value of a function application or not. Therefore, an encoding of abstraction functions requires a *framing axiom* in addition to the definitional axiom, to express the circumstances under which the value of a function application can be framed.

Intuitively, the value of a function can be framed if none of the heap locations on which the function depends are modified. These locations are a subset of the

[3] A formal treatment of these issues is provided by Summers and Drossopoulou [28].
[4] See the online tool [1] for further examples, including a cyclic list.

locations for which the function requires permission in its precondition. However, if the function requires a recursive predicate then this set cannot be determined precisely by the verifier or the SMT solver.

Similarly to existing tools such as Spec# [4] and VeriCool [26], we handle this problem by abstracting over the locations folded inside a predicate instance, via *versioning*. The idea is as follows: if we can be sure that a predicate instance has been neither unfolded nor exhaled since an earlier program point, we know that all locations nested inside the predicate are unmodified. We label the predicate with the same version to identify this case.

Predicate versions are recorded as the values of predicate locations in the heap, which are treated like field locations: in particular, we retain knowledge of a predicate version so long as we hold either direct or known-folded permission to the predicate location. When we hold neither direct nor known-folded permission to such a location, it will naturally be havoced during an exhale. In addition, the version is havoced when the predicate is unfolded. Thus, our solution for function framing is closely tied to the framing of locations.

The details of our handling of functions are given in Sec. 5, but the framing axiom is, informally, as follows: two applications of the same function in two states evaluate to the same value if the receiver and all arguments of the function applications are the same and the two states agree on the values of all locations to which the function precondition requires direct permission; in particular, this includes the versions of the required predicate instances.

To illustrate our approach, consider again two distinct `List` objects `x` and `y` for which the `valid` predicate is held at the beginning in the following code:

```
var i: int := x.itemAt(0);
var j: int := y.itemAt(0);
call y.set(0,10);
assert x.itemAt(0) = i; // succeeds
assert y.itemAt(0) = j; // correctly fails to verify
```

The exhale operation of the call to `y.set` gives away `y.valid`, thereby havocing the version of that predicate. The predicate `x.valid` is not affected, and keeps the same version, which allows the prover to correctly verify the first assertion, using the framing axiom. The second assertion is not necessarily true, and indeed fails to verify because the version of `y.valid` has changed.

4 Encoding of Abstract Predicates

In this and the next section, we present an encoding of our solution in the verification condition generator Boogie [18]. Verification with Boogie consists of three steps: (1) a *translator* translates the source program and its specification into the Boogie language, (2) Boogie computes verification conditions, and (3) an SMT solver attempts to prove the verification conditions. Here, we focus on how the translator encodes abstract predicates without giving recursive definitions or axioms to the prover. The complementary encoding of abstraction functions will be presented in the next section.

Heaps and Permission Masks. Our encoding represents the current heap with a variable `Heap`, which is a map from locations to values. We use the notation (o, l) to refer to the location l of object o; its value in the map `Heap` is denoted by $\texttt{Heap}[o, l]$. Permissions are tracked using *permission masks*, which are sets of locations. The variable `Mask` stores the current mask, which represents the direct permissions held by the current method.

We also store information about predicate instances in the heap. For an abstract predicate p, we use (ghost) predicate locations (o, p) to store a record. Such a record contains the predicate version (as an integer) along with a mask representing the known-folded permissions under the predicate instance $o.p$; we call this the *predicate mask*. We write $\texttt{Heap}[o, p].\texttt{vrs}$ to denote the version of predicate instance $o.p$, and $\texttt{Heap}[o, p].\texttt{msk}$ to denote the corresponding predicate mask for known-folded permissions. At the beginning of verifying a method body, we assume all predicate masks to not contain permissions, that is, to be the empty set.

For two heaps H and H', and for a mask M, we say that H' *preserves* H *according to* M, written $H' \overset{M}{\longleftarrow} H$, if H and H' agree on the location values for which direct or known-folded permission are held in the state described by (H, M), and for all other predicate locations, the predicate mask in H' is empty. More precisely, $H' \overset{M}{\longleftarrow} H$ if, for all locations (o, l):

1. If either $(o, l) \in M$ or $\exists\ (o', p') \in M$ such that $(o, l) \in H[o', p'].\texttt{msk}$, then: $H'[o, l] = H[o, l]$.
2. Otherwise (when the criteria for the previous point to apply do not hold), if $l = p$ for some predicate name p (i.e., (o, l) is a predicate location), then: $H'[o, p].\texttt{msk} = \emptyset$.

An important property of our encoding is that the known-folded permissions of a predicate instance $o.p$ include the known-folded permissions of predicates instances in the body of $o.p$ (informally, the predicate masks record information transitively). This property is maintained as part of our encoding of a **`fold`** statement, and when evaluating an **`unfolding`** expression. Storing transitive information in this "flattened" form means we never need to recursively traverse predicate masks in our encoding.

Encoding of Exhale. As explained in Sec. 2, the exhale operation aggressively havocs the heap and preserves information only for those locations to which the method holds direct or known-folded permission after the exhale. In the encoding of `exhale` (see top of Fig. 3), this is reflected by introducing a fresh heap H', assuming that H' is a framed heap for the state after the exhale and then making H' the new current heap. The actual exhaling is encoded via an auxiliary operation **exhale**$'$, which is explained next.

The **exhale**$'$ operation recursively traverses the assertion to be exhaled, asserting all logical properties, and removing the required permissions from the current mask (see Fig. 3). Exhaling a boolean expression amounts to asserting that the expression holds. Exhaling a conjunction results in exhaling the two

$$[\![\textbf{exhale } A]\!] = \textbf{var } \mathtt{H}' \ ; \ \textbf{havoc } \mathtt{H}' \ ; \ [\![\textbf{exhale}' \ A]\!] \ ; \ \textbf{assume } \mathtt{H}' \overset{\mathtt{Mask}}{\longleftarrow} \mathtt{Heap} \ ; \ \mathtt{Heap} := \mathtt{H}'$$

$$\begin{array}{ll}
[\![\textbf{exhale}' \ e]\!] & = \textbf{assert } \lfloor\!\lfloor e \rfloor\!\rfloor \\
[\![\textbf{exhale}' \ A_1 \ \&\& \ A_2]\!] & = [\![\textbf{exhale}' \ A_1]\!] \ ; \ [\![\textbf{exhale}' \ A_2]\!] \\
[\![\textbf{exhale}' \ e \Rightarrow A]\!] & = \textbf{if } (\lfloor\!\lfloor e \rfloor\!\rfloor) \ \{ \ [\![\textbf{exhale}' \ A]\!] \ \} \\
[\![\textbf{exhale}' \ \textbf{acc}(e.f)]\!] & = \textbf{assert } (\lfloor\!\lfloor e \rfloor\!\rfloor, f) \in \mathtt{Mask} \ ; \ \mathtt{Mask} := \mathtt{Mask} \setminus \{(\lfloor\!\lfloor e \rfloor\!\rfloor, f)\} \\
[\![\textbf{exhale}' \ \textbf{acc}(e.p)]\!] & = \textbf{assert } (\lfloor\!\lfloor e \rfloor\!\rfloor, p) \in \mathtt{Mask} \ ; \ \mathtt{Mask} := \mathtt{Mask} \setminus \{(\lfloor\!\lfloor e \rfloor\!\rfloor, p)\} \\
[\![\textbf{exhale}' \ \textbf{unfolding } e.p \ \textbf{in } e']\!] & = [\![\textbf{inhale } \mathtt{body}(\lfloor\!\lfloor e \rfloor\!\rfloor, p)]\!]^{\mathtt{false}}_{\mathtt{Heap}[\lfloor\!\lfloor e \rfloor\!\rfloor, p].\mathtt{msk}} \ ; \\
 & \quad \textbf{assert } \lfloor\!\lfloor e' \rfloor\!\rfloor
\end{array}$$

Fig. 3. Encoding of exhale. We use A to denote general assertions, which may include permissions, and e to denote expressions without any permissions. To emphasize the uniform treatment of field permissions and predicates, we write $\textbf{acc}(e.p)$ to denote the predicate instance $e.p$, but simply write $e.p$ in our examples for brevity. $[\![\_]\!]$ denotes the translation of source statements to Boogie instructions. In the encoding, we treat **exhale** and $\textbf{exhale}'$ like statements even though they cannot occur in source programs. $\lfloor\!\lfloor \_ \rfloor\!\rfloor$ denotes the translation of source expressions to Boogie expressions; it is straightforward and therefore omitted. Both translation functions refer to the global variables `Heap` and `Mask`. `body`() yields the declared body of a predicate instance.

conjuncts sequentially; the side-effects of these exhales are accumulated. Implications are handled via if-statements in the Boogie output. Exhaling permission to a field or predicate location amounts to checking that the permission is currently held, and then removing it. Exhaling an **unfolding** expression is the most complicated case. First, the body of the unfolded predicate is inhaled, but using the predicate mask for the predicate instance instead of the `Mask` variable. This has the dual effect of assuming any logical properties from the body of the predicate, and recording the permissions encountered as known-folded (but not direct) permissions. The translation of **inhale** used here is described below. Finally, the body of the **unfolding** expression is asserted to be true.

Encoding of Inhale. The translation of **inhale** is given in Fig. 4. In contrast to **exhale**, the translation of **inhale** is parameterised by a mask because it sometimes operates on primary and sometimes on known-folded permissions stored in a particular predicate mask. The boolean parameter b of the translation function indicates whether the inhale is of direct permissions or not. **inhale** traverses the assertion to be inhaled, assuming all logical properties, and adding the required permissions to the current mask. The assumption of $(\lfloor\!\lfloor e \rfloor\!\rfloor, f) \notin \mathtt{M}$ in the case of inhaling field permissions encodes the fact that we cannot hold permission to the same location twice. This assumption is not made for permissions to predicate locations, since it is possible to hold the same predicate more than once[5]. Additionally, when inhaling *known-folded* permission to a predicate $e.p$, all known-folded permissions from the predicate mask for $e.p$ are added to the mask

[5] In this paper, this can happen only for trivial predicates without permissions, but the assumption is important once fractional permissions are employed.

$$
\begin{array}{ll}
[\![\textbf{inhale } e]\!]^b_{\mathtt{M}} & = \textbf{assume } [\![e]\!] \\
[\![\textbf{inhale } A_1 \ \&\& \ A_2]\!]^b_{\mathtt{M}} & = [\![\textbf{inhale } A_1]\!]^b_{\mathtt{M}} \ ; \ [\![\textbf{inhale } A_2]\!]^b_{\mathtt{M}} \\
[\![\textbf{inhale } e \Rightarrow A]\!]^b_{\mathtt{M}} & = \textbf{if } ([\![e]\!]) \ \{ \ [\![\textbf{inhale } A]\!]^b_{\mathtt{M}} \ \} \\
[\![\textbf{inhale acc}(e.f)]\!]^b_{\mathtt{M}} & = \textbf{assume } ([\![e]\!], f) \notin \mathtt{M} \ ; \ \mathtt{M} := \mathtt{M} \cup \{([\![e]\!], f)\} \\
[\![\textbf{inhale acc}(e.p)]\!]^b_{\mathtt{M}} & = \mathtt{M} := \mathtt{M} \cup \{([\![e]\!], p)\} \ ; \\
 & \quad \textbf{\#if } (\neg b) \ \{ \ \mathtt{M} := \mathtt{M} \ \cup \ \mathtt{Heap}[[\![e]\!], p].\mathtt{msk} \ \} \\
[\![\textbf{inhale unfolding } e.p \textbf{ in } e']\!]^b_{\mathtt{M}} & = \textbf{\#if } (b) \ \{ \ [\![\textbf{inhale body}([\![e]\!], p)]\!]^{\mathtt{false}}_{\mathtt{Heap}[[\![e]\!], p].\mathtt{msk}} \ ; \ \} \\
 & \quad \textbf{assume } [\![e']\!]
\end{array}
$$

Fig. 4. Encoding of inhale. The **#if** -conditionals are resolved by the translator.

being used for the inhale. In this way, we maintain the invariant that known-folded permissions are transitively closed; that is, if $(o, p) \in \mathtt{Heap}[o', p'].\mathtt{msk}$ and $(o'', l'') \in \mathtt{Heap}[o, p].\mathtt{msk}$ then $(o'', l'') \in \mathtt{Heap}[o', p'].\mathtt{msk}$.

When exhaling or inhaling an **unfolding** expression for a predicate instance $e.p$, the permissions in the body of $e.p$ become known-folded permissions. Therefore, they get inhaled into the predicate mask of $e.p$. In this case (which is indicated by b being false), inhaling a predicate permission $e.p$ "copies" any known-folded permissions from the inner predicate instance to the outer; that is, we flatten the known-folded permissions under $e.p$. Due to this flattening, an **unfolding** expression affects permissions only when we are inhaling direct permissions (that is, b is true). If we are already inhaling known-folded permissions, any permissions folded under an inner predicate instance are already part of the known-folded masks of the outer predicates.

Encoding of Source Statements. Chalice statements are generally desugared into appropriate combinations of **inhale** and **exhale** statements. For example, a (void) method call is simply encoded with an **exhale** of the precondition followed by an **inhale** of the postcondition (see Fig. 5). The havocing of heap locations that takes place as part of the exhale takes care of invalidating any information that the called method may have changed.

fold and **unfold** statements do not require a general havoc of the heap, since no permissions are actually released to another method; they are simply reorganisations of permissions amongst predicate instances. Therefore, their translations use **exhale**′. In addition to swapping a predicate instance with its body, **fold** statements record known-folded permissions, and **unfold** statements havoc the version of the predicate instance.

We verify loops as usual by using a loop invariant and verifying the loop body independently of the surrounding code. To access locations in the loop body, the verifier requires the appropriate permissions in the loop invariant. To communicate knowledge about known-folded locations between the loop body and the surrounding context, one can use an **unfolding** expression in the invariant. Our encoding of **unfolding** then ensures that the necessary known-folded permissions are added when the loop invariant is evaluated.

$$[\![\textbf{call } e.m()]\!] = [\![\textbf{exhale } \texttt{pre}([\![e]\!], m)]\!] \; ; \; [\![\textbf{inhale } \texttt{post}([\![e]\!], m)]\!]^{\texttt{true}}_{\texttt{Mask}}$$

$$[\![\textbf{fold } e.p]\!] = [\![\textbf{exhale}' \; \texttt{body}([\![e]\!], p)]\!] \; ; \; [\![\textbf{inhale } \texttt{body}([\![e]\!], p)]\!]^{\texttt{false}}_{\texttt{Heap}[[\![e]\!], p].\texttt{msk}} \; ; \; [\![\textbf{inhale } \textbf{acc}(e.p)]\!]^{\texttt{true}}_{\texttt{Mask}}$$

$$[\![\textbf{unfold } e.p]\!] = [\![\textbf{exhale}' \; \textbf{acc}(e.p)]\!] \; ; \; \textbf{havoc } \texttt{Heap}[[\![e]\!], p].\texttt{vrs} \; ; \; \texttt{Heap}[[\![e]\!], p].\texttt{msk} := \emptyset \; ; \; [\![\textbf{inhale } \texttt{body}([\![e]\!], p)]\!]^{\texttt{true}}_{\texttt{Mask}}$$

Fig. 5. Encoding calls, **fold**, and **unfold** statements. `pre`() and `post`() yield the precondition and postcondition of a method with the appropriate substitutions.

5 Encoding of Abstraction Functions

In this section, we present our encoding of abstraction functions, building upon the handling of predicates and known-folded permissions in the previous section. The presented approach is based on uninterpreted functions and axioms, but again avoids presenting the prover with recursive axioms that can be applied in an unbounded way. This is achieved by a combination of versioning predicate instances, and careful selection of axiom triggering strategies [9].

5.1 Function Definitional Axioms and Triggers

Each Chalice function is represented by a corresponding uninterpreted function in the generated Boogie program, in which the heap, the receiver as well as parameters to the Chalice function are turned into explicit parameters. For example, function `length()` from Fig. 1 gives rise to the following uninterpreted Boogie function declaration (by convention, we prepend a `#` symbol to the function name, to differentiate it from its Chalice counterpart):

```
function #length(heap: HeapType, this: ref) returns (int);
```

In Boogie, it is standard to specify properties of an uninterpreted function via *quantified axioms*, e.g., defining the value of a function application, for all states. We call such an axiom (providing the definition of a function) a *definitional axiom*. In the common case that this axiom mentions further (recursive) function applications, the prover needs a strategy to decide when to instantiate the definitional axiom. For example, consider the "direct" translation of the Chalice definition of function `length()`, as a Boogie axiom (we assume here that `H`, `M`, and `this` range over heaps, masks, and references, respectively):

```
axiom ∀ H, M, this • (this,valid) ∈ M ⟹
  #length(H,this) = (H[this,next] ≠ null ? (1 + #length(H,H[this,next])) : 1)
```

This axiom states that, provided that the function's precondition holds, a function application is always equal to its body. Note that Boogie allows multiple quantified variables to be introduced together under one universal quantification. Allowing the prover to instantiate these quantifiers in arbitrary ways, would lead

to two problems. First, the prover might instantiate the axiom with a heap and a mask that belong to different execution states. Second, since the function is recursive, the prover might instantiate its definitional axiom indefinitely in a matching loop.

To solve the first problem, our encoding introduces a boolean uninterpreted function `state` which takes a heap and a mask as arguments. The function application `state(Heap, Mask)` is assumed to yield true in our encoding every time a state is changed, for instance, after an exhale is translated (but not for the intermediate states during the translation). All axioms that quantify over a heap and mask use the `state` function as a premise. We write ∀ `(H,M) • P(H, M)` to abbreviate ∀ `H,M • state(H, M)` ⇒ `P(H, M)`.

To solve the second problem, we make use of Z3's and Boogie's facility to associate universal quantifiers with sets of syntactic triggers. A *trigger set* is a set of terms (the *triggers*), written `{t1,t2,...}`, which can be associated with a universal quantification in Boogie by placing it just before the body of the quantified formula. Trigger sets do not affect the logical meaning of the formula, but prescribe a strategy for controlling the instantiation of the quantifiers introduced. The rule enforced is that the prover must have encountered (somewhere in its proof search) terms matching all of the forms `{t1,t2,...}`, before a corresponding instantiation of the quantified formula can be made. Multiple trigger sets can be provided; in this case, only one set of terms need be fully matched. For example, an axiom of the form

```
∀ x •{f(x),g(x)} {f(g(x))} f(x) = f(g(x))
```

will allow instantiations `f(t)=f(g(t))` to be generated only for terms `t` such that (1) both the terms `f(t)` and `g(t)` have already been encountered by the prover or (2) the term `f(g(t))` has already been encountered.

In the following, we explain how we encode abstraction functions and use triggers in a way that allows the prover to obtain sufficient information from the recursive definitions of the functions, without having the possibility of entering a matching loop.

Limited and Unlimited Functions. To avoid matching loops in the definitional axioms for abstraction functions, we adopt a technique employed in other tools [19]. For each Chalice function we introduce *two* Boogie functions (called the *limited* and *unlimited functions*). Their logical meanings are intuitively the same, but their practical use in axioms and triggers is different. For example, for the `length()` function, we introduce the limited form (which we identify by adding a `'` to the name) along with the original definition:

```
function #length(heap: HeapType, this: ref) returns (int);
function #length'(heap: HeapType, this: ref) returns (int);
```

Now, we define the definitional axiom above as follows: every occurrence of a function application that comes from the body of the function definition, is replaced by its corresponding limited function. The unlimited form of the function

is used in the trigger set for the axiom (and is still used in the translation of source-level Chalice expressions). For example, for `length` we generate the following definitional axiom (all our axioms that quantify over both the heap and mask include `state(H,M)` as part of each trigger, but we omit this for conciseness):

```
axiom ∀ (H,M), this • {#length(H,this)} (this,valid) ∈ M ⟹
  #length(H,this) = (H[this,next] ≠ null ? (1 + #length'(H,H[this,next])) : 1)
```

Because the body of this axiom does not introduce any new applications of the unlimited function `#length()`, an instantiation of the axiom does not give rise to any further instantiations; the potential matching loop is avoided. In order to give a meaning to the `#length'()` function, the following addition axiom is used, which also does not introduce any new applications of the unlimited function `#length()`.

```
axiom ∀ (H,M), this • {#length(H,this)} #length'(H, this) = #length(H, this)
```

Effectively, this allows to prover to unroll a function's definition exactly once for any given occurrence of that function in the source program.

Controlled Triggering. To implement the idea presented in Sec. 3.1 of allowing the prover to unroll a function definition when the corresponding predicate has been folded or unfolded at some point in the program, we proceed as follows. We introduce a boolean function for every predicate, to be used as a trigger for functions that depend on it. We illustrate this using the predicate `valid`:

```
function #valid^trig(this: ref) returns (bool);
```

This function indicates that the corresponding predicate has been folded or unfolded in *some* state for the given receiver. This is introduced in our encoding by instrumenting the translation of **fold** `e.valid` and **unfold** `e.valid` with an extra assumption of $\texttt{\#valid}^{\texttt{trig}}\texttt{(e)}$.

Additionally, we are interested in unrolling the definitional axiom for a function application with a given list of arguments only if that application for the same arguments has been mentioned somewhere in the program. Otherwise, the prover cannot learn useful information by expanding the function's definition. To this end, we add another boolean function along with an axiom:

```
function #length^trig(this: ref) returns (bool);
axiom ∀ (H,M), this • {#length'(H, this)} #length^trig(this)
```

We use the function application $\texttt{\#length}^{\texttt{trig}}\texttt{(e)}$ in triggers to indicate that `length` has been applied to the receiver `e` in *some* state. The axiom shown encodes this meaning.

We now add an additional trigger set to our definitional axiom for `length`. It allows the prover to instantiate the definitional axiom in the cases described in Sec. 3.1, but still does not cause matching loops. Note that this trigger corresponds to our "key insight" in the introduction since it allows the prover to expand recursive definitions up to the depth at which the program has inspected the data structure. The resulting (and final) definitional axiom, is:

```
axiom ∀ (H,M), this • {#length(H,this)} {#length^trig(this), #valid^trig(this)}
 (this,valid) ∈ M ⟹
  #length(H,this) = (H[this,next] ≠ null ? (1 + #length'(H,H[this,next])) : 1)
```

In the general case, a trigger set such as the second one above is introduced for each predicate in the precondition of the function that also gets unfolded in the body of the function before any recursive function calls that occur. If the predicate is not unfolded, or the recursive calls are not in the body of the corresponding **unfolding** expression, the recursion of the function is not tied to traversing the predicate over the data structure, and the additional trigger is not added.

5.2 Function Framing Axiom

To frame functions, we employ an additional axiom which essentially states that, if no part of the state mentioned in a function's precondition differs between two heaps, then the function's value is also the same in the two heaps. As we discussed in Sec. 3.3, we use predicate versions to abstract over the locations folded into the predicate instance, thus avoiding giving recursive predicate definitions to the prover. For example, the framing axiom for `length()` is as follows:

```
axiom ∀ (H1, M1), (H2, M2), this •{#length'(H1,this),#length'(H2,this)}
  H1[this,valid].version = H2[this,valid].version ⟹
    #length'(H1, this) = #length'(H2, this)
```

This axiom is phrased in terms of the limited function `#length'`, but by the axiom relating unlimited and limited functions presented in the previous subsection, it can also be used to frame unlimited functions.

Note that our use of predicate versioning is asymmetrical; if a predicate version is the same in two heaps, then we assume the contents of the predicate instance to be the same, but not vice versa. For example, unfolding and folding a predicate (without modifying any contents) will yield a new version (see Fig. 5). However, our function definitional axiom will be triggered in this case, deducing the relationship to the locations immediately contained inside the predicate instance. If these are known to be preserved, then the prover can conclude that the function value has not changed.

6 Soundness

In this section, we give an informal justification for the soundness of our approach. As we have explained in Sec. 3.2, the definitional axioms for the Boogie functions that we use in our encoding are consistent. The soundness arguments in this section furthermore justify (1) our approach to function framing, and (2) our approach to heap location framing.

6.1 Soundness of Function Framing

To justify our approach to function framing, we use the following definition:

Definition 1 (Permission and Heap Footprints). *The* permission footprint of a predicate *in a heap is the set of heap locations defined by recursively evaluating the predicate definition in the given heap, and collecting the locations whose permissions are required by the definition.*

The permission footprint of a function *in a heap is the set of heap locations defined by evaluating the function's precondition in the given heap, and collecting the locations whose permissions the precondition requires, as well as the permission footprints of all predicates the precondition requires in that evaluation.*

The heap footprint of the predicate or function *in a heap is the set of location-value pairs such that the location is in the corresponding permission footprint, and the value is the value stored in the heap at that location.*

It is sufficient to observe the following:

(1) Evaluation of a function application (at runtime) reads only locations in its permission footprint. This property is enforced by a well-formedness check for each function definition, as is standard for logics supporting user-supplied definitions. In particular, a function application's value is a function of its receiver, arguments, and its heap footprint. Function bodies can also apply functions, but the checking of preconditions ensures that heap footprints for recursive applications are always subsets of the original one.

(2) Like functions, predicate definitions are checked to ensure that they read only heap locations that fall within their permission footprint. Thus, the permission and heap footprints of a predicate are fixed by the permissions folded inside of it. Since we havoc version numbers whenever a predicate is unfolded or exhaled, the version of a predicate location at two different program points can be known to be the same only if the heap footprint of the predicate is also identical at both points.

(3) Consider a situation in which our framing axiom allows the prover to equate a function value between two program points. By (1), we know that it is sufficient to know that the function's heap footprint remains the same between both points. The heap footprint is made up of locations to which explicit permission is required in the function's precondition (which the axiom requires to have the same values), and the heap footprints of those predicate locations that the function's precondition requires (which the axiom requires to have the same versions). By (2), the function value is the same in both states.

6.2 Soundness of Heap Location Framing

For the soundness of heap location framing, the argument depends on a precise definition of the notions of folded and known-folded locations for predicate instances. We address here the soundness of our use of predicate masks to record the known-folded permissions per predicate instance. Our encoding maintains

an invariant regarding these masks whose intuition is simple: we record and remember known-folded permissions until the corresponding predicate instances are lost, and make sure they are recorded in the masks of *all* predicate instances that enclose the locations. It is also important for our argument that we do not selectively record the known-folded permissions from a predicate's body, but always record everything that the body depends on at a time.

We need to define several notions to explain our argument. Since many of our definitions treat both types of location uniformly, we use the meta variable l to range over both predicate and field location names.

Definition 2 (Folded Locations).

- *In a heap H, a location (o', l') is* directly folded *inside a predicate location (o, p), written $directFolded(o, p, o', l', H)$, if evaluating the body of the predicate instance $o.p$ in H results in directly requiring permission to the location (o', l').*
- *In a heap H, a location (o', l') is* folded *inside a predicate location (o, p), written $folded(o, p, o', l', H)$, as defined by (the least fixpoint of):*

$$folded(o, p, o', l', H) \;\Leftrightarrow\; (directFolded(o, p, o', l', H) \lor \exists o'', p''.(directFolded(o, p, o'', p'', H) \;\land\; folded(o'', p'', o', l', H)))$$

We now provide the definitions which characterise the auxiliary information that our encoding records about known-folded permissions.

Definition 3 (Recorded Predicate Bodies and Known-Folded Permissions).

- *In a heap H, a predicate location (o, p) has its* body recorded*, as defined by:*

$$bodyRecorded(H, o, p) \;\Leftrightarrow\; (\forall o', l'.(directFolded(o, p, o', l', H) \;\Rightarrow\; (o', l') \in H[o, p].\texttt{msk} \land (l' \text{ is a predicate location} \;\Rightarrow\; H[o', l'].\texttt{msk} \subseteq H[o, p].\texttt{msk}))$$

- *In a heap H, a location (o', l') is* known-folded *inside a predicate location (o, p), written $knownFolded(o, p, o', l', H)$, as defined by (the least fixpoint of):*

$$knownFolded(o, p, o', l', H) \;\Leftrightarrow\; bodyRecorded(H, o, p) \land (directFolded(o, p, o', l', H) \lor \exists o'', p''.(directFolded(o, p, o'', p'', H) \;\land\; knownFolded(o'', p'', o', l', H)))$$

Our definition of *bodyRecorded* requires not only that *every* location directly required by a predicate body is stored in the corresponding predicate mask, but also that the recorded information is transitive; any information in predicate masks for nested predicate instances must be included in the level above. Our definition of *knownFolded* insists on this organisation of the information in predicate masks "all the way down"—all of the predicate instances in between must also satisfy *bodyRecorded*. This definition of known-folded locations approximates the folded locations for a predicate instance (the definitions are similar, but *knownFolded* enforces extra constraints).

One of the properties we assume about the underlying methodology, is that locations to which folded permission is held, never have their permissions also in the direct mask. This is a special case of the more-general property that permissions should never be forged/duplicated, but only transferred. Note that this is a soundness property of the underlying semantic model.

In the following we argue soundness of our particular encoding. Therefore, we use the two variables `Heap` and `Mask` to refer to the heap and mask, respectively, at a given location in the program.

Lemma 1 (Folded Locations cannot be in the Direct Mask). *Before and after every Chalice program statement, it holds that*

$$\forall o, p, o', l'.\ ((o, p) \in \texttt{Mask} \ \wedge\ \mathit{folded}(o, p, o', l', \texttt{Heap})) \ \Rightarrow\ (o', l') \notin \texttt{Mask}$$

Finally, we can state the invariant that describes how the information in our predicate masks relates to the definition of *knownFolded* locations above:

Theorem 1 (Predicate Mask Permissions are Known-Folded Locations). *Our encoding preserves the following invariant:*

$$\begin{aligned}&\forall o, p, o', l'.((o', l') \in \texttt{Heap}[o, p].\texttt{msk} \ \Rightarrow\\ &\quad(((o, p) \in \texttt{Mask} \ \vee\ \exists o'', p''.((o'', p'') \in \texttt{Mask} \wedge\\ &\quad\mathit{knownFolded}(o'', p'', o, p, \texttt{Heap}))) \ \wedge\ \mathit{knownFolded}(o, p, o', l', \texttt{Heap})))\end{aligned}$$

Proof Sketch. We show that the property is preserved across the four most relevant operations concerning known-folded permissions: folding, unfolding, inhaling, and exhaling predicate instances. For each, we assume the invariant holds beforehand, and show that it holds afterwards. To do this, we consider the cases in which the location (o', l') can newly have become a member of $\texttt{Heap}[o, p].\texttt{msk}$ (in which case the consequent of the implication must also be checked), as well as the cases in which the consequent of the implication may have been falsified (in which case we must be sure that the antecedent is also now false).

Folding $o_1.p_1$: Consider the set of locations directly required in the body of $o_1.p_1$. Permissions to all of these locations are removed from `Mask` and added to $\texttt{Heap}[o_1, p_1].\texttt{msk}$. Furthermore, any of these locations which are predicate locations have their predicate masks added to $\texttt{Heap}[o_1, p_1].\texttt{msk}$ (cf. Fig. 3). In particular, these operations result in $\mathit{bodyRecorded}(\texttt{Heap}, o_1, p_1)$ holding. Finally, (o_1, p_1) is added to `Mask`. Since these operations only add to predicate masks, they do not falsify any previous instances of *knownFolded*.

Considering the invariant, the antecedent $(o', l') \in \texttt{Heap}[o, p].\texttt{msk}$ can newly have been made true only in the case $o = o_1$, $p = p_1$ and for (o', l') being one of the locations directly folded in the body of $o_1.p_1$. For all such cases, we have $\mathit{knownFolded}(o_1, p_1, o', l', \texttt{Heap})$ as required. On the other hand, since a set of locations is removed from `Mask`, we must also take care that the antecedent is not falsified when (o, p) is one of those locations. However, since all such locations are contained in $\texttt{Heap}[o_1, p_1].\texttt{msk}$ by the operation, the second disjunct can be shown in these cases, taking $o'' = o_1$ and $p'' = p_1$.

Unfolding $o_1.p_1$: This operation adds (o_1, p_1) to `Mask`, and removes any information associated with the predicate instance. In particular, $\texttt{Heap}[o_1, p_1].\texttt{msk}$ is set to the empty set $\emptyset$. Since we do not add to any predicate masks, we cannot make the antecedent of the invariant true in any new cases. Since we only remove locations from the predicate mask $\texttt{Heap}[o_1, p_1].\texttt{msk}$, the only instances of *knownFolded* that can be falsified by this are those concerning locations known-folded in (o_1, p_1), i.e., those (o_2, l_2) for which $\mathit{knownFolded}(o_1, p_1, o_2, l_2, \texttt{Heap})$ holds (by Lemma 1, we know that (o_1, p_1) was not itself folded inside any predicate instance). Thus, considering the consequent of the invariant, the only cases we need worry about are when either $o = o_1$ and $p = p_1$ (in which case, the antecedent of the invariant is necessarily false, since we reset the predicate mask to $\emptyset$), or $o'' = o_1$ and $p'' = p_1$. In this latter case, unrolling the definition of $\mathit{knownFolded}(o'', p'', o, p, \texttt{Heap})$ in the state before the operation, in combination with the fact that we record all direct-folded locations from the body of $o_1.p_1$ in the `Mask`, provides sufficient information to show that the invariant still holds.

Inhaling $o_1.p_1$: This operation simply adds (o_1, p_1) to `Mask`, which cannot falsify the consequent of the invariant, or make the antecedent newly true.

Exhaling $o_1.p_1$: This operation removes (o_1, p_1) from `Mask`, and then generates a "global havoc", constrained by the assumption $\texttt{H}' \overset{\texttt{Mask}}{\longleftarrow} \texttt{Heap}$. Let's consider the point just after this havoc operation (see Fig. 3), but before the new heap $\texttt{H}'$ is assigned to `Heap` (so that we have names for both the new and old heaps). The havoc operation means that all predicate masks $\texttt{H}'[o.p].\texttt{msk}$ are set to $\emptyset$, except in the case that either $(o, p) \in \texttt{Mask}$ still holds after the operation, or, for some (o'', p''), $(o'', p'') \in \texttt{Mask} \wedge (o, p) \in \texttt{Heap}[o'', p''].\texttt{msk}$ holds. In particular, consider the cases in which the antecedent of the invariant $(o', l') \in \texttt{H}'[o, p].\texttt{msk}$ could possibly hold. This would require that $\texttt{H}'[o, p].\texttt{msk} \neq \emptyset$, and, since nothing is added to the predicate masks in the operation, also that $(o', l') \in \texttt{Heap}[o, p].\texttt{msk}$ held. By the argument above, along with the assumption of the invariant in the state before the operation, we deduce that:

$$
\begin{array}{l}
(o', l') \in \texttt{H}'[o, p].\texttt{msk} \;\Rightarrow \\
\quad (((o, p) \in \texttt{Mask} \;\vee\; \exists o'', p''.((o'', p'') \in \texttt{Mask} \wedge \\
\quad \mathit{knownFolded}(o'', p'', o, p, \texttt{Heap})) \;\wedge \mathit{knownFolded}(o, p, o', l', \texttt{Heap})))
\end{array}
$$

Thus, all we need to know in order to deduce that the invariant holds in the new heap $\texttt{H}'$ is that the occurrences of *knownFolded* mentioned here are still true for $\texttt{H}'$. This follows because, in both cases, the known-folded information is recorded under a predicate instance to which direct permission is held. By the soundness of the underlying permission logic, we know that the locations folded inside this outer predicate instance cannot be modified, and thus, that the meanings of all nested predicates are preserved. Furthermore, any predicate instances known to be folded inside this outer instance must have their entire bodies recorded in the corresponding predicate mask, and therefore, their meanings and directly folded locations remain unaffected by the global havoc. Therefore, a simple induction shows that *knownFolded* information is preserved in the new heap. □

Corollary 1 (Predicate Masks record only Folded Permissions). *Before and after the translation of every Chalice statement, the following property holds:*

$$\forall o, p, o', l'.\ ((o,p) \in \texttt{Mask} \ \wedge\ (o',l') \in \texttt{Heap}[o,p].\texttt{msk} \ \Rightarrow\ \mathit{folded}(o,p,o',l',\texttt{Heap}))$$

By this corollary, which follows directly from Theorem 1, the locations framed across a global havoc are always either locations to which direct permission is held, or which are folded inside a predicate instance to which direct permission is held; in both cases, the soundness of the underlying permission handling implies that framing these location values is sound.

7 Related Work

The concept of abstract predicates [23] is used in both separation logic [24] and implicit dynamic frames [27]. In terms of VCG verifiers, abstraction functions are used along with abstract predicates in Chalice [20] and VeriCool [27].

The VeriCool VCG implementation handles abstract predicates and abstraction functions soundly but, unlike our solution, introduces the possibility of matching loops in the SMT solver. To assess whether matching loops are a problem in practice, we took an example from VeriCool and translated it into Chalice. We experimented with partial specifications by leaving out parts of the main loop invariant. We found that the VeriCool verification time significantly increased for failed proof attempts, from 2.3 minutes to up to one hour (at which point we terminated the verifier). In some cases, removing logically redundant parts of the specification also increased the verification time (to 16 minutes). These are typical symptoms of matching loops. Our new version of Chalice verifies the translated program in less than 20 seconds (with or without the redundant specifications) and reports failed attempts for partially specified versions even faster. Details about this experiment can be found at [1]. In particular, the site contains the code in both languages and marks the conjuncts of the VeriCool invariant according to how their removal affected the verification attempt and what the corresponding behavior of Chalice was.

The previous Chalice encoding of predicates and functions is unsound. The encoding considered only direct permissions and havoced the heap *lazily*, that is, when permissions are (re-)obtained. This ensures that values of fields are preserved, but also leaves invalid information in the heap, which caused the unsound behaviour. In particular, folded locations were never havoced.

Symbolic execution is an alternative technique to VCG and is used in tools such as [5,11,15,16,26]. Typically, symbolic execution engines use partial heaps and other more elaborate data structures for the representation of the program state. In the presence of such data structures, the problem treated in this paper is not as intricate. In particular, symbolic execution engines can iterate through their heap representation to determine folded permissions, whereas for VCG with SMT solvers, this is not possible in the presence of recursive predicates. Symbolic execution forgets heap information by chopping off the corresponding part from

the partial heap. Framing of function values can be sufficiently handled as in VCG by predicate versioning.

The mechanism of predicate versioning typically seen in symbolic execution engines differs from ours. A version in these systems is a snapshot of the underlying heap. If a predicate is unfolded and folded back immediately, its version does not change and functions depending on that predicate are known to not have changed their value either. In contrast, our approach always changes the version number at **unfold** statements. However, the use of the definitional axiom allows the prover to unroll the definition one level and prove function equivalence nonetheless. In this way, we can achieve the same effect as in the predicate versioning of symbolic execution, without using the symbolic execution-specific data structures, which are unsuitable in a VCG setting. Versioning in Spec# [4] is also similar to our predicate versioning, but more incomplete: the user must explicitly mention the functions whose return values must be preserved.

Bardou's PhD thesis [3] presents a verification technique that uses hierarchies of memory regions. To improve performance, the encoding into the Why intermediate language flattens these hierarchies into a number of separate heaps. For recursive regions, a complete flattening would result in an unbounded number of heaps. Therefore, Bardou's encoding determines statically how deeply to flatten such hierarchies, based on the access paths in a method and a fixed depth limit. Our approach requires neither such a static analysis nor a fixed limit because **unfold** statements determine when to expand a predicate definition, and triggers control where the prover may expand a function definition.

Madhusudan et al. [22] present a logic to express complex properties of tree structures, and a procedure that decides these efficiently. Their core idea of expanding a recursive definition a statically known number of times that depends on the program under verification is similar to ours. Compared to our work, their logic is restricted in expressiveness: it tackles only tree structures and it considers only functions with a single tree argument and a specific definition pattern.

Shape Analysis has been successfully applied in the context of three-valued logic [25], and more recently adapted to separation logic [10], with the aim of *inferring* the recursive structure of the current heap. The approach is typically based on a fixed set of recursive definitions, but some techniques (e.g., [13]) aim also to infer these definitions. The idea that recursive definitions need only be handled up to a certain depth is central to shape analysis, but the problem tackled is different; we do not aim at such inference, while we do support arbitrary user-defined predicates and dependent abstraction functions.

Suter et al. provide a generic approach to constructing decision procedures for recursive algebraic data types [29]. In particular, their work supports recursive abstraction functions, to allow a more abstract representation of the underlying data to be used in specifications. While their approach applies only to functional data types, they employ a notion of partial evaluation of the abstraction functions that is similar to our controlled instantiation of function definitions. It would be interesting to see if their work, as well as work on shape analyses could be adapted to our setting, perhaps to infer **unfold** and **fold** statements.

8 Conclusion

In this paper, we have presented a VCG encoding technique for abstract predicates and abstraction functions. To prevent matching loops one must refrain from giving recursive definitions to the prover, even though in general the definitions of both predicates and functions can be recursive. We solve this challenge with the insight that proof obligations can typically be discharged by allowing the prover to unroll recursive definitions for the parts of the program data structures which have been observed by the program at some earlier point. Inspecting the program text allows us to encode this with the use of trigger strategies as well as the introduction of known-folded permissions.

Our encoding is, to the best of our knowledge, the only sound encoding of both features that prevents matching loops. Our comparison with the VeriCool VCG verifier shows that prevention of matching loops makes an important difference in the verification experience.

We have implemented the methodology for the more general setting of fractional permissions [6] in a new version of Chalice[6] [2]. We ran our implementation on the Chalice test suite of 100 interesting examples and regression tests and observed no unsoundness or incompleteness. The timings are predictable, even for examples with faulty specifications. Our tool can also be tried out online [1], where we also provide several challenging examples.

Acknowledgements. We thank Jan Smans for explaining details of the VeriCool tool, and Malte Schwerhoff for discussions and putting our tool online. We thank Sophia Drossopoulou for early feedback, and the anonymous reviewers for their detailed suggestions. We are grateful to the reviewers of FM 2012 for their feedback on an earlier version of this paper, which led to a major redesign and improvement of our technique.

References

1. Chalice (online), `http://boogiebox2.inf.ethz.ch:1001/tuwin/tool/chalice`
2. Chalice source code repository, `http://chalice.codeplex.com`
3. Bardou, R.: Verification of Pointer Programs Using Regions and Permissions. PhD thesis, Université de Paris-Sud 11 (2011)
4. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. CACM 54(6), 81–91 (2011)
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
6. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)

[6] Our implementation has been successfully evaluated by the ECOOP artifact evaluation committee, who confirmed that it met their expectations.

7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
8. Darvas, Á., Müller, P.: Reasoning about method calls in interface specifications. JOT 5(5), 59–85 (2006)
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
10. 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)
11. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA, pp. 213–226. ACM (2008)
12. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013)
13. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: PLDI, pp. 256–265. ACM (2007)
14. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1(4), 271–281 (1972)
15. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
16. Kassios, I.T., Müller, P., Schwerhoff, M.: Comparing verification condition generation with symbolic execution: an experience report. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 196–208. Springer, Heidelberg (2012)
17. Leino, K.R.M.: Specification and verification of object-oriented software. In: *Marktoberdorf International Summer School 2008. Lecture Notes* (2008)
18. Leino, K.R.M.: This is Boogie 2. Working Draft (2008), `http://research.microsoft.com/en-us/um/people/leino/papers.html`
19. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: SAC, pp. 615–622. ACM (2009)
20. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
21. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD V 2008/2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
22. Madhusudan, P., Qiu, X., Stefanescu, A.: Recursive proofs for inductive tree data-structures. In: POPL, pp. 123–136. ACM (2012)
23. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL, pp. 247–258. ACM (2005)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. IEEE Computer Society Press (2002)
25. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118. ACM (1999)
26. Smans, J., Jacobs, B., Piessens, F.: VeriCool: An automatic verifier for a concurrent object-oriented language. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 220–239. Springer, Heidelberg (2008)

27. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
28. Summers, A.J., Drossopoulou, S.: A formal semantics for isorecursive and equirecursive state abstractions. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 129–153. Springer, Heidelberg (2013)
29. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL, pp. 199–210. ACM (2010)

Really Automatic Scalable Object-Oriented Reengineering

Marco Trudel[1], Carlo A. Furia[1], Martin Nordio[1], and Bertrand Meyer[1,2]

[1] Chair of Software Engineering, ETH Zurich, Switzerland
[2] Software Engineering Laboratory, ITMO, St. Petersburg, Russia
firstname.lastname@inf.ethz.ch

Abstract. Even when implemented in a purely procedural programming language, properly designed programs possess elements of good design that are expressible through object-oriented constructs and concepts. For example, placing structured types and the procedures operating on them together in the same module achieves a weak form of encapsulation that reduces inter-module coupling. This paper presents a novel technique, and a supporting tool AutoOO, that extracts such implicit design elements from C applications and uses them to build reengineered object-oriented programs. The technique is completely automatic: users only provide a source C program, and the tool produces an object-oriented application written in Eiffel with the same input/output behavior as the source. An extensive evaluation on 10 open-source programs (including the editor `vim` and the math library `libgsl`) demonstrates that our technique works on applications of significant size and builds reengineered programs exhibiting elements of good object-oriented design, such as low coupling and high cohesion of classes, and proper encapsulation. The reengineered programs also leverage advanced features such as inheritance, contracts, and exceptions to achieve a better usability and a clearer design. The tool AutoOO is freely available for download.

1 Introduction

The reasons behind the widespread adoption of object-oriented programming languages have to be found in the powerful mechanisms they provide, which help design and implement clear, robust, flexible, and maintainable programs. Classes, for example, are modular constructs that support strong encapsulation, which makes for components with high cohesion and low coupling; inheritance and polymorphism make classes extensible, thus promoting flexible reuse of implementations; exceptions can handle inter-procedural behavior without polluting functional and modular decomposition; and contracts seamlessly integrate specification and code, and support abstract yet expressive designs.

Competent programmers, however, try to achieve the same design goals—encapsulation, extensibility, and so on—even when they are implementing in a programming language that does not offer object-oriented features. A developer adopting the C programming language, for example, will use files as primitive modules collecting **struct**s and functions operating on them; will implement exception handling through a disciplined use of *setjmp* and *longjmp*; will use conditional checks and defensive programming to define valid calling contexts in a way somewhat similar to preconditions.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 477–501, 2013.

Following these observations, this paper describes work to automatically reengineer procedural C programs to introduce object-oriented features, based on design elements such as files, function signatures, and user-defined types. The result of our work is a fully automated technique and supporting tool that extract such *implicit* design information from C programs[1] and use it to *reengineer* functionally equivalent *object-oriented* applications in the Eiffel object-oriented programming language.

Given the huge availability of high-quality C applications, an automatic technique to reengineer C into object-oriented code has a major potential practical impact: reusing legacy code in modern environments. In fact, this is not the first attempt at supporting object-oriented reengineering, and porting procedural applications to a modern programming paradigm is a recurrent industrial practice. A careful analysis of related work, which we present in Section 8, shows however that previous approaches have limitations in terms of comprehensiveness, automation, applicability to real code, and achieved quality of the reengineering. In contrast, our approach constitutes a significant contribution with the following distinguishing characteristics.

- The reengineering technique is *fully automatic* and implemented in the freely available tool AutoOO. Users only need to provide an input C project; AutoOO outputs an object-oriented Eiffel application that can be compiled.
- The technique and tool work on real software of considerable size, as demonstrated by an extensive evaluation on 10 open-source programs including the editor `vim` and the math library `libgsl`.
- As demonstrated by quantitative analysis of the products of the automatic reengineering, the object-oriented code achieves good encapsulation and introduces inheritance, contracts, and exceptions when feasible.
- The reengineering is correct by construction: the generated object-oriented programs achieve the same functional behavior as the source programs and do not introduce potentially incorrect refactorings that might break the code.

These characteristics make AutoOO a valuable asset to reuse good-quality software in object-oriented environments. We have experienced the usefulness of this service firsthand with the Eiffel user community, which is not as large as those of other mainstream languages, and hence it lacks a wide choice of libraries in some application domains. Prompted by numerous requests, in addition to the 10 programs discussed in Section 5, we used AutoOO to port some C libraries that were sorely needed by the Eiffel developer's community: the driver for the MongoDB database; the PCRE regular expression library; and the SDL mixer audio library. After being produced with minimal effort, the Eiffel versions of these libraries are now being used by Eiffel developers. Requests for converting more libraries keep coming, and AutoOO is starting to be directly used by programmers other than its authors. This gives us confidence that our work is practical and helps solve a real and recurrent problem: automatic and scalable reuse.

While AutoOO translates C to Eiffel, the principles and reengineering techniques it implements are based on standard object-oriented features, and hence are readily applicable to other programming languages—such as C++, Java, and C#—offering classes,

[1] We target ANSI C and GCC extensions.

static members, visibility modifiers, and exceptions.[2] In fact, to highlight the generality of the reengineering, the presentation will use a Java-like syntax; this will be palatable to readers familiar only with C-based programming languages without misrepresenting any conceptually relevant aspect. We assume knowledge of the standard terminology and notions of object-oriented programming [15].

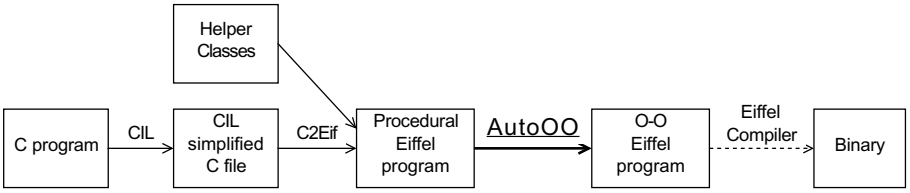


Fig. 1. Object-oriented reengineering with C2Eif and AutoOO

C2Eif and AutoOO. The reengineering techniques described in this paper are combined with our previous work on C2Eif [27] and implemented as part of the toolchain shown in Figure 1. The toolchain implements an overall transformation that inputs a C program and outputs an object-oriented Eiffel project with the same functionality. The C source program is first processed by CIL [19], which simplifies some C constructs (for example, there is only one type of loop in CIL). In the second stage, C2Eif transliterates the CIL output into procedural Eiffel, whose structure replicates that of the C input program without introducing elements of object-oriented design. We described this stage in previous work [27]; its details are largely independent of the reengineering techniques implemented by AutoOO, which only uses C2Eif as a back-end. Finally, AutoOO processes the C2Eif output, introduces the transformations described in Sections 3 and 4, and outputs the reengineered object-oriented Eiffel programs that can be compiled.

Tool Availability. AutoOO is available online at `http://se.inf.ethz.ch/research/c2eif`. The webpage includes AutoOO's sources, pre-compiled binaries, source and binaries of all translated programs of Table 2, and a user guide. *AutoOO's distribution has been successfully evaluated by the ECOOP artifact evaluation committee and found to meet expectations.* For ease of presentation, we will use the name AutoOO to denote not only the tool but also the reengineering technique it implements.[3]

[2] The only two features used by AutoOO that may not be universally available are contracts and member renaming during inheritance. Contracts, however, are increasingly provided in other languages as libraries or assertions (e.g., CodeContracts for C#, **assert** in Java). Member renaming plays a limited role in the refactorings produced by AutoOO (Section 3.4), and a translator targeting another language could make up for it by using the same name in the super- and subclasses, or even (as we suggest in Section 3.4) by dropping inheritance in the few cases where renaming is required.

[3] In the latest distributions, C2Eif and AutoOO are integrated into a single translator (called C2Eif for simplicity), which offers the option to apply the object-oriented reengineering transformations presented in this paper.

Outline. In the rest of the paper, Section 2 defines the goals of AutoOO reengineering, how they are assessed, and the design principles followed. Section 3 discusses how AutoOO introduces elements of object-oriented design—in particular, how it populates classes. Section 4 discusses how it introduces contracts and exceptions. Section 5 presents the evaluation of the correctness, scalability, and performance of AutoOO based on 10 reengineered applications and libraries. Section 6 reviews the fundamental aspects of the object-oriented design style introduced by AutoOO and how they make for usable reengineered programs. Section 7 discusses the current limitations of AutoOO. Section 8 reviews related work and compares AutoOO against existing tools and approaches to object-oriented reengineering. Section 9 concludes and outlines future work.

2 O-O Reengineering: Goals, Principles, and Evaluation

The overall goal of AutoOO reengineering is *expressing* the design *implicit* in procedural programs using constructs and properties of the object-oriented paradigm. For example, we restructure and encapsulate the code into classes that achieve a high cohesion and low coupling, we make use of inheritance to reuse code, and so on. The main motivation for introducing object-oriented constructs is that they can *explicitly* express design and structure of programs concisely and in a way amenable to further flexible extension and reuse.

While reengineering in its most general meaning—the reconstruction of "a system in a new form" [2]—may also introduce new functionality or mutate the existing one (for example, with corrective maintenance), the present work tries not to deviate from the original intentions of developers as reflected in the procedural implementations.[4] For example, we do not introduce exceptions unless the original program defines some form of inter-procedural execution path. We adopt a conservative approach because we want a reengineering technique that:

- is *completely automatic*, not just a collection of good practices and engineering guidelines;
- always produces *correct* reengineerings, that is programs that are functionally equivalent to the original procedural programs.

Improving and extending software are important tasks, but largely orthogonal to our specific goals and requiring disparate techniques. For example, there are serviceable tools to infer specifications from code (to mention just a few: [5,12,28]) which can be applied atop our reengineering technique to get better code specification automatically; but including them in our work would weaken the main focus of the contribution.

From the user perspective, AutoOO is a translator that takes an input C program and converts it to an object-oriented Eiffel program that replicates its functionality. The rest of this section presents other specific goals of AutoOO and how we assess them.

[4] This entails that, when applied to C programs that do not contain elements conducive to object-oriented design, AutoOO should simply introduce few changes. The experiments with the programs of Table 1 suggest, however, that AutoOO's heuristics are often applicable with success.

Case Studies. The evaluation of AutoOO, described in the following sections, targets 10 open-source programs totalling 750 KLOC. The 10 programs include 7 applications and 3 libraries; all of them are widely-used in Linux and other "*nix" distributions. `hello world` is the only toy application, which is however useful as a baseline. The other applications are: `micro httpd` 12dec2005, a minimal HTTP server; `xeyes` 1.0.1, a widget for the X Windows System that shows two googly eyes following the cursor movements; `less` 382-1, a text terminal pager; `wget` 1.12, a command-line utility to retrieve content from the web; `links` 1.00, a simple web browser; `vim` 7.3, a powerful text editor. The libraries are: `libcurl` 7.21.2, a URL-based transfer library supporting protocols such as FTP and HTTP; `libgmp` 5.0.1, for arbitrary-precision arithmetic; `libgsl` 1.14, a powerful numerical library. Section 5 discusses more details about the programs used in the experiments.

Correctness, Scalability, and Performance. In addition to systematic interactive usage, we assess *correctness* of the reengineering produced by AutoOO by running the standard regression test-suites available with the programs, hereby verifying that the output is the same in C and Eiffel. We also consider the *translation time* taken by AutoOO to guarantee that it scales up; and the *performance* of the Eiffel reengineered program to ensure that it does not incur a slowdown that severely compromises usability. Section 5 discusses these correctness and performance results.

Object-Oriented Design. AutoOO creates an object-oriented program consisting of a collection of classes; each class aggregates data definitions (fields) and functions operating on them (methods). Section 3 presents the technique that extracts object-oriented design; we evaluate the quality of the object-oriented design produced by AutoOO with the following measures:

Soundness: We manually inspected 43% of all classes produced by AutoOO (all projects but `vim` and `libgsl`) and we determined how many methods belong to the correct class, that is are indeed methods operating on the fields of the class.[5]

Coupling and Cohesion: The coupling of a class is measured as the ratio: number of accesses to members of other classes / number of accesses to members of the same class. When this ratio is low (less than 1 in the best cases), it shows that classes are loosely coupled and with high cohesion.[6]

Information Hiding Is measured as the ratio of private to public members. A high ratio indicates that classes make good usage of information hiding for encapsulation.

Instance vs. Class Members: The ratio of instance to class members (called **static** members in Java) gives an idea of the "object-orientedness" of a design. A high ratio indicates a really *object* oriented design, as it makes limited usage of "global" class fields and methods.

Inheritance: We manually inspected all uses of inheritance introduced by AutoOO and we determined how many correctly define substitutable heir classes.

[5] As we illustrate in Section 3, soundness refers to whether reengineering moves members to the "right" classes from a design point of view. Soundness is thus a notion orthogonal to *correctness*: AutoOO reengineerings do not alter behavior and hence are always correct (as per standard regression testsuites and general usage).

[6] Cohesion is normally defined as the dual of coupling.

Contracts and Exceptions. In addition to the core elements of object-oriented design, AutoOO also introduces high-level features often present in object-oriented languages: contracts and exceptions. AutoOO clearly distinguishes the purpose of contracts vs. exceptions.

Contracts. Replace annotations (not part of ANSI C but available as GCC extensions) and encode simple requirements on a function's input and guarantees on its output; they are discussed in Section 4.1.

Exceptions. Replicate the behavior of *setjmp* and *longjmp* which divert the structured control flow in exceptional cases across functions and modules; they are discussed in Section 4.2.

3 Object-Oriented Design

Table 1. Object-oriented design metrics after each reengineering step applied to the ten case study programs

REENGINEERING STEP	# bundle methods	# datatype methods	# bundle fields	# datatype fields	% sound datatype methods	average coupling	overall hiding	instance/class members	# inheriting classes
1. source files	12,445	0 (0%)	3,628	5,337 (60%)	–	8.87 1.54	–	0.33	0
2. function signature	7,724	4,721 (38%)	3,628	5,337 (60%)	94%	2.33 1.20	–	0.88	0
3. call graph	6,471	5,974 (47%)	2,881	6,084 (68%)	96%	2.00 1.06	0.12	1.12	0
4. inheritance	6,471	5,974 (47%)	2,881	6,084 (68%)	96%	2.00 1.06	0.12	1.12	4

AutoOO produces object-oriented designs that consist of collections of classes. The generated classes are of two kinds with different purposes:

- a *datatype class* combines the data definitions translating some C type definition (**struct** or **union**) with a collection of instance methods translating C functions operating on the type.
- a *bundle class* collects global variables and global functions present in some C source file and makes them available to clients as class members.

Only datatype classes are germane to object-oriented design, which emphasizes proper encapsulation of data definitions with the operations defined on them; bundle classes, however, are still necessary to collect elements that do not clearly belong exclusively to any datatype, such as globals shared by multiple clients. Thus, bundle classes are a safe fall-back that keeps the original modular units (the source files) instead of forcing potentially unsound refactorings.

Corresponding to their roles, datatype classes contain mainly "proper" *instance* members (Section 3.3 discusses the exceptions), whereas bundle classes contain only *class* members (also called **static**).

AutoOO generates datatype and bundle classes in four steps:

1. *Source file* analysis creates the bundle classes and populates them based on the content of source files; it creates a datatype class for each structured type definition (**struct** or **union**).
2. *Function signature* analysis refactors methods from bundle to datatype classes, moving operations closer to the data definition they work on.
3. *Call graph* analysis refactors members from bundle to datatype classes, and shuffles methods among datatype classes, moving members to classes where they are exclusively used.
4. *Inheritance* analysis creates inheritance relationships between datatype classes based on their fields.

The following subsections 3.1–3.4 describe the steps in detail with examples.

Table 1 reports how the various metrics mentioned in Section 2 change as we apply the four steps to the 10 case study programs. For each reengineering step, Table 1 reports:

- The number of bundle and datatype members[7] created, partitioned in methods and fields.
- The percentage of *sound* datatype methods.[8] A method *m* of a datatype class *T*—that contains the data definition of a **struct** *T* or **union** *T*—is *sound* if manual analysis confirms that *m* implements an operation whose primary purpose is modifying or querying instances of *T*.
- The average (median) *coupling* of classes, where the coupling of a class *T* (with respect to the rest of the system) is defined as follows. An *access* is the read or write of a field, or a method call; an access in the body of a method *m* of *T* is *in* if it refers to a member of *T* other than *m*; it is *out* if it refers to a member of a class other than *T*. When counting accesses in a method *m* we ignore duplicates: if *m*'s body calls *r* more than once, we only count it as one access. *T*'s coupling is the ratio of out to in accesses of all its members. For each step, Table 1 reports two values of coupling; the value on top puts all classes of all programs together (hence larger projects dominate), while the bottom value computes medians per programs and then the median across programs.
- The *hiding* of classes, measured as the ratio of **private** and **protected** to **public** members.
- The ratio of *instance* to *class* members.
- The number of classes defined using *inheritance*.

The rest of this section discusses the figures shown in Table 1 to demonstrate how each reengineering step improves these object-oriented design metrics. A word of caution is necessary about the reliability of metrics such as those we use to assess the improvement of design quality, something which eludes general quantitative definitions [3]. Nonetheless, metrics give an idea of how the design changes through the various transformations; while the exact values they report should be taken with a grain of salt, they are still useful to complete the picture of how AutoOO performs in practice.

[7] A bundle (or datatype) member is a member of a bundle (or datatype) class.

[8] Evaluated on all projects but `vim` and `libgsl`, as discussed in Section 2.

3.1 Source File Analysis

For each source file `F.c` in the program, the first reengineering step creates a bundle class *F* and populates it with translations of all the global variables and function definitions found in `F.c`. For each definition of a structured type *T* in `F.c`, the first step also creates a datatype class *T* that contains *T*'s components as fields. AutoOO only has to consider structured type definitions using **struct** or **union**; atomic type definitions and **enum**s are handled in the initial processing by C2Eif. Since AutoOO's reengineering treats the two kinds of structured type declarations uniformly, we only deal with **struct**s in the following to streamline the presentation; the handling of **union**s follows easily.

```
int majority_age = 18;

struct person
{
    int age;
    bool sex;
};

void set_age(struct person *p, int new_age) {
    if(new_age ≤ 0) return;
    p→age = new_age;
}

bool overage(int age) {
    return (age > majority_age);
}

bool is_adult(struct person *p) {
    return overage(p→age);
}
```

Fig. 2. C source file `PersonHandler.c`.

For example, when processing the C source file in Figure 2, the first step generates the datatype class *Person* and the bundle class *PersonHandler* in Figure 3.

Source file analysis sets up the dual bundle/datatype design and defines the classes of the system. The result is still far from good object-oriented design as the datatype classes are just empty containers mapping **struct**s one-to-one, and in fact we have no hiding and a low instance/class member ratio (the only instance members are the datatype fields).

The overall coupling (first row in Table 1) is also quite high after step 1. This does not come as a surprise: because all methods are located in bundle classes, every read or write of a **struct** field from the original C code becomes an out access. The proliferation of out accesses is especially evident in `libgmp`, where the majority of modules have *only*

```
class Person
{
    int age;

    boolean sex;
}
```

```
class PersonHandler
{
    static int majority_age = 18;

    static void set_age(Person p, int new_age) {
        if(new_age≤0) return;
        p.age = new_age;
    }

    static boolean overage(int age) {
        return (age > majority_age);
    }

    static boolean is_adult(Person p) {
        return overage(p.age);
    }
}
```

Fig. 3. Datatype class *Person* (left) and bundle class *PersonHandler* (right) initially created for the C file in Figure 2

out accesses. In general, coupling is higher for libraries in our experiments; this may indicate that coupling for library code should be measured differently, for example by considering the library in connection with a client. In any case, this is not a problem for our evaluation: the value of coupling after step 1 is merely a baseline that corresponds to purely procedural design; our goal is to measure how this value changes as we apply the next reengineering steps.

3.2 Function Signature Analysis

The second reengineering step moves methods from bundle to datatype classes according to their signature, with the intent of having data and methods operating on them in the same class.

Consider a method *m* of bundle class *M* with signature

$$t_0\ m\ (t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ ,$$

for $n \geq 0$. An argument p_k of *m* is *data-bound* if its type $t_k = T*$ (pointer to *T*), where *T* is a datatype class. When a routine has more than one such argument, we consider only the first one in signature order. A data-bound argument p_k is *globally used* by *m* if it is accessed (read or written) at least once along every path of *m*'s control flow graph, except possibly for argument handling paths. An *argument handling path* is a path guarded by a condition that involves some argument p_h, with $h \neq k$, and terminated by a **return**.

For each method *m* of a bundle class *M* that has a data-bound argument p_k of type $t_k = T*$ which is globally used, the second reengineering step moves *m* into the

datatype class *T* and changes its signature—which becomes non-**static** and drops argument p_k—and its body—which refers to p_k implicitly as **this**. Accordingly, *m*'s body may have to adjust other references to members of *M* that are now in a different class; also any call to *m* has to be adjusted following its new signature.

Continuing the example of Figure 2, the second reengineering step determines that *set_age* and *is_adult* can be refactored: argument *p* is data-bound and globally used in both methods (with the first instruction in *set_age* being an argument handling path). Hence, the two methods are moved from class *PersonHandler* to class *Person* which becomes:

```
class Person
{
    int age;

    boolean sex;

    void set_age(int new_age) {
        if(new_age≤0) return;
        age = new_age;
    }

    boolean is_adult() {
        return PersonHandler.overage(age);
    }
}
```

Function signature analysis introduces fundamental elements of object-oriented design. As reported in Table 1, manual inspection reveals that 94% of the methods moved to datatype classes are indeed operations on that type; this means that 97% of the members of datatype classes (fields plus sound methods) are refactored correctly. Remember that our definition of soundness refers to design, not to correct behavior: even the $6\% = 100\% - 94\%$ "unsound" methods behave correctly as in the original C programs, even if they are arguably not allocated to the best class. Inspection also reveals some common causes of unsound refactorings. Some functions use a generic pointer (type **void**$*$) as first argument, and then cast it to a specific **struct**$*$ in the code; and in a few cases the pointer arguments are simply not reliable indicators of data dependence or are in the wrong order (more details below).

Coupling drastically reduces after step 2, because many methods that access fields of datatype classes are now located inside those classes. This dominates over the increase in out accesses to bundle members from within the methods moved to datatype classes, also introduced by step 2. In particular, function signature analysis mitigates the high coupling we measured in the libraries. Finally, many methods have become instance methods, with an overall instance/class ratio of 0.88.

How restrictive is the choice to consider only the first data-bound argument to a datatype class for deciding where to move methods? For example, if the code in Figure 2 had another function **void** *do_birthday*(**struct** *person* $*p$, **struct** *log* $*l$) that increases *p*'s age and writes to the log pointed to by *l*, should we move *do_birthday* to datatype class *log* instead of *person*? The empirical evidence we collected suggests that

our heuristics is generally not restrictive: we manually analyzed all 77 functions with multiple arguments of type "pointer to **struct**" in the case study programs and found only 3 cases where the "sound" refactoring would target an argument other than the first.

Another feature of function signature analysis as it is implemented in AutoOO is the choice to ignore methods with an argument p whose type t corresponds to a **struct**, but that is passed by *copy* (in other words, whose original type in C is t rather than $t*$); we found 131 such cases among the programs of Table 2 and only 56 (43%) of them would have generated a sound refactoring. In all, we preferred not to consider arguments passed by copy because it would lead to unsound refactoring in the majority of cases; a more sophisticated analysis of this aspect belongs to future work.

Finally, the refactoring requirement that a data-bound argument must be globally used is not necessary, in most cases, to achieve soundness, but dropping it would introduce incorrect translations that change the behavior of the program in some cases. In fact, a function with an argument not used globally includes valid executions where the argument is allowed to be **null**; therefore, it cannot become an instance method which always has an implicit non-**null** target **this**.

As an interesting observation about the application of reengineering to the programs of Table 2, we found that 40% of the methods moved to datatype classes in step 2 have a name that includes the datatype class name as prefix. For example, the methods operating on a datatype class *hash_table* in `wget` are named *hash_table_get*, *hash_table_put*, and so on. This suggests that, in the best cases, even purely syntactic information carries significant design choices. AutoOO takes advantage of this finding and removes such prefixes to increase the readability of the created code (see also the client example in Section 6).

3.3 Call Graph Analysis

The third reengineering step moves more members to datatype classes according to where the members are used, with the intent of encapsulating "utility" members together with the datatype definitions that use them exclusively.

Consider a member *n* of any class *N* that is accessed (read, written, or called) *only* in a datatype class *T*. For each such member *n*, the third reengineering step moves *n* into the datatype class *T*. If *n* is an instance method or a class method, it becomes an instance method; if it is a class field, it remains a class field to preserve the original semantics of **static** fields corresponding to global C variables (this is the only case where we add class members to datatype classes). Members moved to datatype classes in this step also become **private**, since they are not used outside the class they are moved to. Since moving a member out of a class changes the global call graph, AutoOO performs the third reengineering step iteratively: it starts with the member *n* with the largest number of accesses, and updates the call graph after every refactoring move, recalculating the set of candidate members for the next move.

Continuing the example of Figure 2, assume that method *overage* of bundle class *PersonHandler* is only called by *is_adult* in datatype class *Person*, and that field *majority_age* is instead read also by other modules. Then, AutoOO moves *overage* to *Person* where it becomes non-**static** and **private**:

```
private boolean overage(int age) {
    return (age > PersonHandler.majority_age);
}
```

The field *majority_age*, instead, stays unchanged in class *PersonHandler*.

As reported in Table 1, call graph analysis refines the object-oriented design and introduces hiding when possible, that is for 12% of the members. Even if there are 2,290 private members, these are localized in only 139 classes, hence the average hiding per class is low (3% mean). Coupling decreases once more, as a result of moving utility methods to the class where they are used. The percentage of sound refactored methods increases to 96%; overall, 98% of the datatype members are refactored correctly. Step 3 also makes instance members the majority (53% of all members, or 1.12 instance member per class member).

Under the conservative approach taken by AutoOO, which creates functionally equivalent code, the values of hiding, coupling, and instance/class members reached after steps 1–3 strike a fairly good balance between introducing object-oriented features and preserving the original design as not to harm understandability due to unsound members in classes of the reengineered application. The example in Section 6 reinforces these conclusions from a user's perspective.

3.4 Inheritance Analysis

The fourth reengineering step introduces inheritance in order to make existing subtyping relationships between datatype classes explicit. In the original C code subtyping surfaces in the form of casts between different **struct** pointer types. Because the language does not provide any way to make one **struct** type conform to another, modelling subtyping in C requires frequent *upcasting* (conversion from a subtype to a supertype) as well as *downcasting* (from a supertype to a subtype). Inheritance analysis finds such casting patterns and establishes inheritance relationships between the involved types.

Consider two type declarations in the source C program:

struct *r* { $t_1\ a_1; t_2\ a_2; \ldots; t_m\ a_m;$ };
struct *s* { $u_1\ b_1; u_2\ b_2; \ldots; u_n\ b_n;$ };

We say that type *s* is *cast* to type *r* if there exists, anywhere in the program's code, a cast of the form (**struct** *r*∗) *e* with *e* an expression of type **struct** *s*∗. We say that type *s extends* type *r* if $n > m$[9] and, for all $1 \leq i \leq m$, the types t_i and u_i are equivalent. For every such types *r* and *s* such that *s* extends *r* and *s* is cast to *r*, *r* is cast to *s*, or both, the fourth reengineering step makes the datatype class for *s* inherit from *r*. Using a **renames** clause[10] to rename fields with different names, *s* becomes:

class *s* **extends** *r* **renames** $a_1{:}b_1, a_2{:}b_2, \ldots, a_m{:}b_n$
{
    $u_{m+1}\ b_{m+1};$
    ...

[9] The case $n = m$ could be also supported but would rarely be useful with the programs tried so far.

[10] Available natively in Eiffel and not in Java, but whose semantics is straightforward.

```
    u_n b_n;
    // Rest of the class unchanged.
}
```

Notice that AutoOO bases inheritance analysis on type information only, not on field *names*. Therefore, it requires renaming of fields in general; implementing this feature in Java or similar languages, where renaming is not possible, would require some workaround (or simply dropping inheritance when renaming is required).

Continuing the example of Figure 2, assume another **struct** declaration is **struct** *student* { **int** *age*; **bool** *sex*; **int** *gpa*; } and that, somewhere in the program, a variable of type *person* $*$ is cast to (*student* $*$). Then, datatype class *Student* becomes:

```
class Student extends Person
{
    int gpa;
    /*...*/
}
```

While AutoOO identified 1,875 pairs t_1, t_2 of types where t_1 extends t_2, and 96 pairs where t_1 is cast to t_2, only 4 pairs satisfy both requirements. Hence, the introduction of inheritance in our experiments is limited to 4 classes (2 in each of `xeyes` and `less`). This is largely a consequence of the original C design where extensions of **struct**s along these lines are infrequent, combined with the constraint that our reengineering create functionally equivalent code and be automatic. All few uses of inheritance AutoOO identified are, however, sound, in that the resulting types are real subtypes that satisfy the substitution principle. In contrast, manual inspection reveals that none of the other $92 = 96 - 4$ pairs of cast types determine classes that are related by inheritance. Introducing inheritance for the other 1,871 pairs solely based on one type extending the other is most likely unsound without additional evidence. Many **struct**s, for example, are collections of integer fields, but they model semantically disparate notions that are not advisable to combine. The other metrics in Table 1 do not change after inheritance analysis, assuming we count fields in the *flattened* classes.

Interestingly, the two instances of inheritance we found in `less` use renaming to define lists as simplified header elements. For example:

```
struct element_list {
    struct element *first;
};
```

```
struct element {
    struct element *next;
    char *content;
};
```

The two types are indeed compatible, and the renaming makes the code easier to understand even without comments.

4 Contracts and Exceptions

AutoOO introduces contracts and exceptions to improve the readability of the classes generated in the reengineering. Section 4.1 explains how AutoOO builds contracts from

compiler-specific function annotations and from simple implicit properties of pointers found by static analysis. Section 4.2 discusses how exceptions can capture the semantics of *longjmp*.

4.1 Contracts

Contracts are simple formal specification elements embedded in the program code that use the same syntax as Boolean expressions and are checked at runtime. AutoOO constructs two common kinds of contracts that annotate methods, namely preconditions and postconditions. A method's precondition (introduced by **requires**) is a predicate that must hold whenever the method is called; it is the caller's responsibility to establish the method's precondition before calling it. A method's postcondition (introduced by **ensures**) is a predicate that must hold whenever the method terminates; it is the method's body responsibility to guarantee the postcondition upon termination.

AutoOO creates contracts from two information sources commonly available in C programs:

- GCC function attributes;
- globally used pointer arguments.

Based on these, AutoOO added 3,773 precondition clauses and 13 postcondition clauses to the programs in Table 2.

GCC Function Attributes. The GCC compiler supports special function annotations with the keyword ***__attribute__***. GCC can use these annotations during static analysis for code optimization and to produce warnings if the attributes are found to be violated. Among the many annotations supported—most of which are relevant only for code optimization, such as whether a function should be inlined—AutoOO constructs preconditions from the attribute ***nonnull*** and postconditions from the attribute ***noreturn***. The former specifies which of a function's arguments are required to be non-**null**; the latter marks functions that never return (for example, the system function *exit*). For each method $m\ (t_1\ p_1, \ldots, t_m\ p_m)$ corresponding to a C function with attribute ***nonnull*** $(i_1, \ldots, i_n)$, with $n \geq 0$ and $1 \leq i_1, \ldots, i_n \leq m$ denoting arguments of *m* by position, AutoOO adds to *m* the precondition

requires $p_{i_1} \neq$ **null**, $p_{i_2} \neq$ **null**, . . ., $p_{i_n} \neq$ **null**

that the arguments $p_{i_1}, \ldots, p_{i_n}$ be non-**null**. For each method *m* corresponding to a C function with attribute ***noreturn***, AutoOO adds to *m* the postcondition **ensures false** that would be violated if *m* ever terminates.

Extending the example of Figure 2, the function:

```
__attribute__ (( nonnull (2), noreturn ))
void kill(struct person *p, struct person *q) {
    /*...*/
    printf("A person is killed at age %d", q→age);
    exit(1);
}
```

gets the following signature after reengineering (assuming the first argument becomes **this**):

void *kill*(*Person q*) **requires** $q \neq$ **null** **ensures false**

GCC function attributes determined 266 precondition and 13 postcondition clauses in the programs of Table 2.

Globally Used Pointers. Section 3.2 defined the notion of *globally used* argument: an argument that is accessed (read or written) at least once along every path in a method's body. Based on the same notions, for each pointer argument p of a method m that is globally used in m on *all* paths (including argument-handling paths), AutoOO adds to m the precondition **requires** $p \neq$ **null** that p be non-**null**. The precondition does not change the behavior of the method: if m were called with $p =$ **null**, m would eventually crash in every execution when accessing a **null** reference, and hence $p \neq$ **null** is a necessary condition for m to correctly execute.

Through globally used pointer analysis, AutoOO introduced 3,507 precondition clauses in the programs of Table 2.

Defensive programming is a programming style that tries to detect violations of implicit preconditions and takes countermeasures to continue execution without crashes. For example, when function *set_age* in Figure 2 is called with a non-positive *new_age*, it returns without changing *p*'s age field, thus avoiding corrupting it with an invalid value. While defensive programming and programming with contracts have similar objectives—defining necessary conditions for correct execution—they achieve them in very different ways: while contracts clearly specify the semantics of interfaces and assign responsibilities for correct execution, defensive programming just tries to communicate failures while working around them. This fundamental difference is the reason why we do not use contracts to replace instances of defensive programming when reengineering: doing so would change the behavior of programs. In the case of *set_age*, for example, a precondition **requires** *new_age* > 0 would cause the program to terminate with an error whenever the precondition is violated, whereas the C implementation continues execution without effects. In addition, C functions often use integer return arguments as error codes to report the outcome of a procedure call; introducing contracts would make clients incapable of accessing those codes in case of error.

Relaxed Contracts for Memory Allocation. The GCC distribution we used in the experiments provides ***__attribute__*** annotations (see Section 4.1) also for system libraries. In particular, the memory allocation functions *memcpy* and *memmove*:

__attribute__ ((***nonnull*** (1, 2)))
extern void ∗*memcpy*(**void** ∗*dest*, **const void** ∗*src*, *size_t n*);

__attribute__ ((***nonnull*** (1, 2)))
extern void ∗*memmove*(**void** ∗*dest*, **const void** ∗*src*, *size_t n*);

require that their pointer arguments *dest* and *src* be non-**null**. By running the reengineering produced by AutoOO, we found that this requirement is often spuriously violated at runtime: when the functions are called with the third argument n equal to 0, they return without accessing either *dest* or *src*, which can therefore safely be **null**. Correspondingly, AutoOO builds the contracts for these functions a bit differently:

requires $n == 0$ || (*dest* $\neq$ **null** && *src* $\neq$ **null**)

that is *dest* and *src* must be non-**null** only if *n* is non-zero. This inconsistency in GCC's annotations does not have direct effects at runtime in C because annotations are not checked. We ignore whether it might have other subtle undesirable consequences as the compiler may use the incorrect information to optimize binaries.

4.2 Exceptions

Object-oriented programming languages normally include dedicated mechanisms for handling exceptional situations that may occur during execution. While error handling is possible also in procedural languages such as C, where it is typically implemented with functions returning special error codes, exceptions in object-oriented languages are more powerful because they can traverse the call stack searching for a suitable handler; this makes it possible to easily cross the method and class boundaries in exceptional situations, without need to introduce a complex design that harms the natural modular decomposition effective in all non-exceptional situations.

C programmers can explicitly implement a similar mechanism that jumps across function boundaries with the library functions *setjmp* (save an arbitrary return point) and *longjmp* (jump back to it). AutoOO detects usages of these library functions and renders them using exceptions in the object-oriented reengineering. AutoOO defines a helper class *CE_EXCEPTION* which can use Eiffel's exception propagation mechanism to go back in the call stack to the allocation frame of the method that called *setjmp*. There, local jump instructions reach the specific point saved with *setjmp* within the method's body. We do not discuss the details of the translation because they refer to several low-level mechanisms discussed in [27] that are out of scope in the present paper. From the point of view of the reengineering, however, the translation expresses the complex semantics of *longjmp* naturally through the familiar exception handling mechanism.

AutoOO found 6 usages of *longjmp* in the programs of Table 2, which it replaced with exceptions.

We did not make a more extensive usage of exceptions, for example for replacing return error codes. In many cases, it would have complicated the object-oriented design and slowed down the program, without significant benefits. A fine-grained analysis of the instances of defensive programming, with the goal of selecting viable candidates that can be usefully translated through exceptions, belongs to future work.

5 Correctness, Scalability, and Performance

In addition to the metrics of object-oriented design displayed in Table 1 and discussed in the previous sections, we evaluated the *behavior* of the reengineering produced by AutoOO on the 10 programs in Table 2. All the experiments ran on a GNU/Linux box (kernel 2.6.37) with a 2.66 GHz Intel dual-core CPU and 8 GB of RAM, GCC 4.5.1, CIL 1.3.7, EiffelStudio 7.0.8.

The reengineering of each program proceeds as previously shown in Figure 1, with the end-to-end process (from C source to object-oriented Eiffel output) being push-button.

Table 2. Reengineering of 10 open-source programs

	SIZE (LOCS) PROCEDURAL (C)	O-O (EIFFEL)	# CLASSES	TRANS-LATION (S)	BINARY SIZE (MB)
`hello world`	8	15	1	1	1.1
`micro httpd`	565	1,983	16	1	1.3
`xeyes`	1,463	10,665	77	1	1.6
`less`	16,955	22,709	75	5	2.3
`wget`	46,528	61,040	178	24	4.1
`links`	70,980	108,726	227	31	12.5
`vim`	276,635	414,988	669	138	22.6
`libcurl`	37,836	70,413	272	17	–
`libgmp`	61,442	82,379	223	20	–
`libgsl`	238,080	378,025	729	81	–
TOTAL	750,492	1,150,943	2,467	319	45.5

For each program used in our evaluation, Table 2 reports: the size of the source procedural program in C (after processing by CIL); the size of the reengineered object-oriented program in Eiffel output by AutoOO; the number of classes generated by the reengineering; the source-to-source time taken by the reengineering (including both C2Eif's translation and AutoOO's reengineering, but excluding compilation of Eiffel output to binary); the size of the binary after compiling the Eiffel output with EiffelStudio[11].

Correctness. In all cases, the output of AutoOO successfully compiles with EiffelStudio without need for any adjustment or modification. After compilation, we ran extensive trials on the compiled reengineered programs to verify that they behave as in their original C version. We performed some standard usage sessions with the interactive applications (`xeyes`, `less`, `links`, and `vim`) and verified that they behave as expected and they are usable interactively. We also performed systematic usability tests for the other applications (`hello world`, `micro httpd`, and `wget`) which can be used for batch processing; and ran standard regression testsuites (also automatically translated from C to Eiffel) on the libraries. All usability and regression tests execute and pass on both the C and the translated Eiffel versions of the programs, with the same logged output.

Scalability of the reengineering process is demonstrated by the moderate translation times (second to last column in Table 2) taken by AutoOO: overall, reengineering 750 KLOC of C code into 1.1 MLOC of Eiffel code took less than six minutes.

Performance. We compared the performance of AutoOO's reengineered output against C2Eif's non-reengineered output for the non-interactive applications and libraries of Table 2. The performance is nearly identical in C2Eif and AutoOO for all programs but the `libgsl` testsuite, which even executed 1.33 times *faster* in the reengineered AutoOO version. This shows that the object-oriented reengineering

[11] In EiffelStudio, libraries cannot be compiled without a client.

produced by AutoOO improves the design without overhead with respect to a bare non-reengineered translation. The basic performance overhead of switching from C to Eiffel—analyzed in detail in [27]—significantly varies with the program type but, even when it is pronounced, it does not preclude the usability of the translated application or library in standard conditions. These conclusions carry over to programs reengineered with AutoOO, and every optimization introduced in the basic translation provided by C2Eif will automatically result, at the end of the tool chain, in faster reengineered applications.

6 Discussion: AutoOO's Object-Oriented Style

All object-oriented designs produced by AutoOO deploy a collection of classes partitioned into bundle and datatype classes, as explained in Section 3. While this prevents a more varied gamut of designs from emerging as a result of the automatic reengineering, in our experience it does not seem to hamper the readability and usability of the reengineered programs, as we now briefly demonstrate with a real-world example. We attribute this largely to the fact that AutoOO produces *sound* reengineering in most cases, and programs with correct behavior in all cases. Therefore, the straightforward output design is understandable by programmers familiar with the application domain, who can naturally extend or modify it to introduce new functionality or a more refined design.

As mentioned in Section 1, we have distributed to the Eiffel developers community a number of widely used C libraries translated with AutoOO. One of them is MongoDB, a document-oriented (non-relational) database[12]. Consider a client application that uses MongoDB's API to open a connection with a database and retrieve and print all documents in a collection *tutorial.people*. Following the API tutorial, this could be written in C as shown in Figure 4 on the left. A client using the MongoDB library translated and reengineered by AutoOO would instead use the syntax shown in Figure 4 on the right.

On the one hand, the two programs in Figure 4 are structurally similar, which entails that users familiar with the C version of MongoDB will have no problem switching to its object-oriented counterpart, and would still be able to understand the C documentation in the new context. On the other hand, the program on the right nicely conforms to the object-oriented idiom: variable definitions are replaced by object creations (lines 2 and 7); and function calls become instance method calls (lines 3, 8, 12, and 17). Method names are even more succinct, because they lose the prefixes "*mongo_*" and "*mongo_cursor_*" unnecessary in the object-oriented version where the type of the target object conveys the same information more clearly.

The only departure from traditional object-oriented style is the call to the cursor destruction function on line 14, which remains a **static** method call with identical signature. AutoOO did not turn it into an instance method because its implementation can be called on **null** pointers, in which case it returns without any effect:

int *mongo_cursor_destroy*(*mongo_cursor* ∗*cursor*) { **if**(!*cursor*) **return** 0; /∗ ... ∗/ }

[12] `http://www.mongodb.org/display/DOCS/C+Language+Center`

```
1  // connect to database
2  mongo conn[1];
3  int status = mongo_connect(conn,
4                  "127.0.0.1", 27017);
5
6  // iterate over database content
7  mongo_cursor cursor[1];
8  mongo_cursor_init(cursor, conn,
9                  "tutorial.people");
10 while(mongo_cursor_next(cursor)
11         == MONGO_OK) {
12    bson_print(&cursor→current);
13 }
14 mongo_cursor_destroy(&cursor);
15
16 // disconnect from database
17 mongo_destroy(conn);
```

```
// connect to database
Mongo conn = new Mongo();
int status = conn.connect(
                  "127.0.0.1", 27017);

// iterate over database content
MongoCursor cursor = new MongoCursor();
cursor.init(conn.address,
                  "tutorial.people");
while(cursor.next() == MONGO_OK)
{
    cursor.current.print();
}
Mongo.mongo_cursor_destroy(cursor.address);

// disconnect from database
conn.destroy();
```

Fig. 4. A MongoDB client application written in C (left) and the same application written for the AutoOO translation of MongoDB (right)

As discussed in Section 3.2, AutoOO does not reengineer such functions because the target of an object-oriented call is not allowed to be **null**. In such cases, users may still decide that it is safe to refactor by hand such examples; in any case, the AutoOO translation provides a proper reengineering of most of the library functionalities.

7 Limitations

By and large, the evaluation with the programs of Table 2 demonstrates that AutoOO is a scalable technique applicable to programs of considerable size and producing good-quality object-oriented designs automatically. This section discusses the few limitations that remain, distinguishing between those of the underlying C to Eiffel translation and those of the object-oriented reengineering.

C to Eiffel Translation. As discussed in detail in [27], the raw translation from C to Eiffel provided by C2Eif does not currently support: a few rare programming patterns that rely on specific memory layouts, such as how the arguments passed to a function are stored next to one another; and a few GCC exotic extensions. Using CIL as preprocessor, while it contributes to simplifying and maintaining the translation, also carries its own limitations: K&R legacy C is not supported; and comments are stripped and formatting is lost, and hence this information cannot be used to improve the readability and formatting of the translated Eiffel code. None of these limitations is intrinsic to the AutoOO approach, and lifting them is largely an engineering effort: we plan to remove the dependency on CIL as well as to support additional non-standard features of the C language if they will be often needed by users of AutoOO.

Object-Oriented Reengineering. All the limitations of our reengineering technique follow the decision to be conservative, that is not to change the behavior in any case, to only extract design information already present in the C programs, and to only introduce

refactorings with empirically demonstrated high success rates, in terms of accurately capturing design elements. For example, Section 3.2 discussed how a refactoring based on struct arguments passed by copy would lead to less than 50% of sound refactorings, while we normally aim at success rates over 90%.

While these requirements make it possible to have a robust and fully automatic technique, they may also be limiting in some specific cases where users are willing to push the reengineering, accepting the risk of having to revise the output of AutoOO before using it.

Formal Correctness Proofs. A final limitations of our work is the lack of formal correctness proofs of the basic C translation and of the reengineering steps. While the evaluation (discussed in Section 5) extensively tested the translated applications without finding any unexpected behavior—which gives us good confidence in the robustness of the results—this still falls short of a fully formal approach such as [4]. This is planned as future work.

8 Related Work

Reengineering [2] is a common practice—and an expensive activity [22]—in professional software development. Given the wide adoption of languages with object-oriented features, object-oriented reengineering is frequently necessary. In this section, we briefly review some general literature on reengineering of legacy systems (8.1), followed by a detailed analysis of significant approaches to object-oriented reengineering (8.2). For lack of space, we do not include a general review of *refactoring* techniques and methods [7], as the focus of this paper is extracting object-oriented designs automatically from the analysis of procedural code rather than refactoring per se.

8.1 Reengineering of Legacy Systems

The main goal in reengineering a legacy system is raising the level of abstraction. Typically, this is achieved by *translating* an implementation written in an old programming

Table 3. Tools translating C to object-oriented languages

	target language	completely automatic	available	readability	external libraries	pointer arithmetic	gotos	inlined assembly
`Ephedra` [14]	Java	no	no	+	no	no	no	no
`C2J++` [26]	Java	no	no	+	no	no	no	no
`C2J` [21]	Java	no	yes	−	no	yes	no	no
`C++2Java` [25]	Java	no	yes	+	no	no	no	no
`C++2C#` [25]	C#	no	yes	+	no	no	no	no
AutoOO	Eiffel	yes	yes	+	yes	yes	yes	yes

language—such as K&R C, Fortran-77, or old COBOL—into a modern programming language such as Java [16,1,29]. This process does not normally include improving the object-oriented design but only making the same system available in a supported environment.

To summarize the state-of-the art in this area, Table 3 lists five tools that translate C to an object-oriented language *without object-oriented reengineering* and compares them against AutoOO. The table is taken from our previous work on C2Eif [27], the C to Eiffel translator on top of which we built AutoOO—which therefore appears as last entry of the table. For each tool, Table 3 reports (see [27] for more details):

- The *target language*.
- Whether the tool is *completely automatic*, that is whether it generates translations that are ready for compilation without need for any manual rewrite or adaptation.
- Whether the tool is *available* for download and usable.
- An assessment of the *readability* of the code produced.
- Whether the tool supports unrestricted calls to *external libraries*, unrestricted *pointer arithmetic*, unrestricted **goto**s, and inlined *assembly code*.

Even at the level of bare translation of C programs without object-oriented reengineering, the currently available tools do not support the full C language used in real programs because they cannot translate features such as external libraries and unrestricted pointer arithmetic, whose exact behavior is very complicated to get right but is necessary to have fully automatic translation tools. A recent comparative evaluation covering a wide range of tools for legacy system reengineering [18] points to similar limitations that prevent achieving complete automation. AutoOO, in contrast, can count on C2Eif's full support of the complete C language used in real programs, which underpins the development of a robust and scalable object-oriented reengineering tool.

8.2 Object-Oriented Reengineering

Among the broad literature on reengineering for modern systems, we identified nine approaches that target specifically object orientation. Table 4 summarizes their main features and compares them with ours. Following the primary goals of our work, described in Section 2, Table 4 lists:

- The *source* and the *target* languages (or if it is a generic methodology).
- Whether *tool support* was developed, that is whether there exists a tool or the paper explicitly mentions the implementation of a tool. A YES in small caps denotes the only currently publicly available tool, namely AutoOO.
- Whether the approach is *completely automatic*, that is if it performs O-O reengineering without any user input other than providing a source procedural program.
- Whether the approach supports the *full* source *language* or only a subset thereof.
- Whether the approach has been *evaluated*, that is whether the paper mentions evidence, such as a case study, that the approach was tried on real programs. If available, the table indicates the size of the programs used in the evaluation.
- Whether the approach performs *class identification*, that is if it groups fields and methods in classes.

– Whether the reengineering technique introduces object-oriented features, namely it identifies *instance methods* (as opposed to class methods which should have a restricted role in object orientation) and uses of *inheritance*.

Table 5 gives some notes about *limitations* of the approaches.

Table 4. Comparison of approaches to O-O reengineering

	source–target	tool support	completely automatic	full language	evaluated	class identification	instance methods	inheritance
Gall [9]	methodology	no	no	–	yes	yes	?	no
Jacobson [10]	methodology	no	no	–	yes	yes	no	–
Livadas [13]	C–C++	yes	no	no	no	yes	yes	no
Kontogiannis [11]	C–C++	yes	no	?	10KL	yes	?	yes
Frakes [8]	C–C++	yes	no	no	2KL	yes	?	no
Fanta [6]	C++–C++	yes	no	no	120KL	yes	?	no
Newcomb [20]	Cobol–OOSM	yes	yes	no	168KL	yes	?	no
Mossienko [17]	Cobol–Java	yes	no	no	25KL	yes	no	no
Sneed [23]	Cobol–Java	yes	yes	no	200KL	yes	?	no
Sneed [24]	PL/I–Java	yes	yes	no	10KL	yes	?	no
AutoOO	C–Eiffel	YES	yes	yes	750KL	yes	yes	yes

Table 5. Overview of limitations

	LIMITATIONS
Gall [9]	requires assistance of human expert
Jacobson [10]	only defines a process; 3 case studies from industry
Livadas [13]	prototype implementation; no support for pointers
Kontogiannis [11]	sound reengineering for only about 36% of the source code
Frakes [8]	translation may change the behavior; requires expert judgement
Fanta [6]	requires expert judgement
Newcomb [20]	only a model is generated, no program code
Mossienko [17]	only partial automation; the translation may change the behavior
Sneed [23]	domain-specific translation
Sneed [24]	domain-specific translation

[20] and [23] are the only authors that report evaluations on code bases of significant size. [20]'s reengineering, however, produces OOSM (hierarchical object-oriented state machine models) models; mapping OOSM to a standard compilable object-oriented language is not covered. [23] reports that some manual corrections of the automatically generated Java code were necessary during the translation of the code base, although these manual interventions were later implemented as an extension of the translator;

anyway, [23] targets the translation of domain-specific applications and in fact does not support the full input language; the authors of [23] expect that tackling new applications will require extending the tool.

[11]'s approach to introduce inheritance is based on the analysis of **struct** fields and of function signatures. AutoOO also uses **struct** field analysis to introduce inheritance, but limits the analysis to field types and ignores field names (see Section 3.4).

A direct detailed comparison of other tools with AutoOO on specific object-oriented features is difficult to obtain as several of these works focus on some aspects of the reengineering but provide few concrete details about other aspects or about how the reengineering is performed on real code. This is also the reason for the presence of "?" in the column "instance methods", corresponding to cases where we could not figure out the details of how methods are refactored. In light of the evidence collected (or lack thereof), it is fair to say that identifying instance methods automatically without expert judgement is an open challenge; and so are full source language support and complete automation. These features are a novel contribution of AutoOO.

9 Conclusions and Future Work

We presented a new completely automatic approach to object-oriented reengineering of C programs and a freely available supporting tool AutoOO. AutoOO scales to applications and libraries of significant size, and produces reengineered object-oriented programs that are directly compilable and usable with the same behavior as the source C programs. The reengineered object-oriented designs produced by AutoOO encapsulate fields and methods operating on them with a high degree of soundness—thus lowering coupling and increasing cohesion—and make judicious usage of inheritance, contracts, and exceptions to improve the quality of the object-oriented design.

Future Work. The remaining limitations of AutoOO, discussed in Section 7, suggest items for future work:

- When we recognize the usage of standard library services (e.g., generic data structures), we will replace them with their Eiffel counterparts when possible, extending existing approaches for mapping APIs [30].
- Section 3 discussed possible additional sources of information to improve the amount of sound reengineered methods in datatype classes (e.g., **struct** arguments passed by copy). The empirical data we collected in our experiments suggest, however, that these sources would frequently lead to unsound refactorings if directly applied. For these cases, we will investigate more sophisticated analyses that try to understand in which cases a sound refactoring is possible.
- Finally, we will explore the possibility of combining AutoOO's completely automatic approach with additional user input (e.g., domain knowledge useful to understand the software design), with the goal of tailoring the reenginering to each application.

Acknowledgments. This work was partially supported by the ETH grant "Object-oriented reengineering environment".

References

1. Achee, B.L., Carver, D.L.: Creating object-oriented designs from legacy FORTRAN code. JSS 39(2), 179–194 (1997)
2. Chikofsky, E.J., Cross II, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software 7(1), 13–17 (1990)
3. Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., Moghadam, I.H.: Experimental assessment of software metrics using automated refactoring. In: ESEM, pp. 49–58. ACM (2012)
4. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: POPL, pp. 533–544 (2012)
5. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE TSE 27(2), 99–123 (2001)
6. Fanta, R., Rajlich, V.: Reengineering object-oriented code. In: Proceedings of the International Conference on Software Maintenance, pp. 238–246 (1998)
7. Fowler, M.: Refactoring: Improving the design of existing code. Addison-Wesley (1999)
8. Frakes, W.B., Kulczycki, G., Moodliar, N.: An empirical comparison of methods for reengineering procedural software systems to object-oriented systems. In: Mei, H. (ed.) ICSR 2008. LNCS, vol. 5030, pp. 376–389. Springer, Heidelberg (2008)
9. Gall, H., Klosch, R.: Finding objects in procedural programs: an alternative approach. In: WCRE, pp. 208–216. IEEE (1995)
10. Jacobson, I., Lindström, F.: Reengineering of old systems to an object-oriented architecture. In: OOPSLA, pp. 340–350. ACM (1991)
11. Kontogiannis, K., Patil, P.: Evidence driven object identification in procedural code. In: STEP, pp. 12–21. IEEE (1999)
12. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
13. Livadas, P.E., Johnson, T.: A new approach to finding objects in programs. Journal of Software Maintenance 6(5), 249–260 (1994)
14. Martin, J., Müller, H.A.: Strategies for migration from C to Java. In: CSMR, pp. 200–210. IEEE Computer Society (2001)
15. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall (1997)
16. Millham, R.: An investigation: reengineering sequential procedure-driven software into object-oriented event-driven software through UML diagrams. In: COMPSAC, 2002, pp. 731–733 (2002)
17. Mossienko, M.: Automated Cobol to Java recycling. In: CSMR, pp. 40–50. IEEE (2003)
18. Nadera, B.S., Chitraprasad, D., Chandra, V.S.S.: The varying faces of a program transformation systems. ACM Inroads 3(1), 49–55 (2012)
19. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
20. Newcomb, P., Kotik, G.: Reengineering procedural into object-oriented systems. In: WCRE, pp. 237–249. IEEE (1995)
21. Novosoft. C2J: a C to Java translator (2001), `http://www.novosoft-us.com/solutions/product_c2j.shtml`
22. Sneed, H.: Planning the reengineering of legacy systems. IEEE Software 12(1), 24–34 (1995)
23. Sneed, H.: Migrating from COBOL to Java. In: ICSM, pp. 1–7. IEEE (2010)
24. Sneed, H.: Migrating PL/I code to Java. In: CSMR, pp. 287–296. IEEE (2011)
25. Tangible Software Solutions. C++ to C# and C++ to Java, `http://www.tangiblesoftwaresolutions.com/`

26. Tilevich, E.: Translating C++ to Java. In: German Java Developers' Conference Journal. Sun Microsystems Press (1997)
27. Trudel, M., Furia, C.A., Nordio, M., Meyer, B., Oriol, M.: C to O-O translation: Beyond the easy stuff. In: Proceedings of WCRE, pp. 19–28. IEEE (2012)
28. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: ICSE, pp. 191–200. ACM (2011)
29. Yeh, A., Harris, D., Reubenstein, H.: Recovering abstract data types and object instances from a conventional procedural language. In: WCRE, pp. 227–236 (1995)
30. Zhong, H., Thummalapenta, S., Xie, T., Zhang, L., Wang, Q.: Mining API mapping for language migration. In: ICSE, pp. 195–204 (2010)

Detecting Refactored Clones

Mati Shomrat[1] and Yishai A. Feldman[2]

[1] Blavatnik School of Computer Science, Tel Aviv University
matis@post.tau.ac.il
[2] IBM Research – Haifa
yishai@il.ibm.com

Abstract. The availability of automated refactoring tools in modern development environments allows programmers to refactor their code with ease. Such tools, however, enable developers to inadvertently create code clones that quickly diverge in form but not in meaning. Furthermore, in the hands of those looking to confuse plagiarism-detection tools, automated refactoring may be abused to avoid discovery of copied code.

We present Cider, an algorithm that can detect code clones regardless of various refactorings that may have been applied to some of the copies but not to others. Most significant is the ability to discover interprocedural clones, where parts of one copy have been extracted to separate methods. We evaluated Cider on several open-source Java projects, attempting to detect interprocedural clones between successive versions of each project. Interprocedural clones were detected in all evaluated projects, demonstrating the pervasive nature of the problem. Compared to a manual assessment, Cider performed well in terms of both recall and precision.

1 Introduction

Code duplication occurs for a variety of reasons [1]. Once duplicated, each copy takes on a life of its own, as different transformations are applied to it. Some of these transformations may be behavior-preserving (that is, refactorings [2,3]), while other may modify the behavior of a copy. Such modifications may be due to genuine differences between the ways the copies are used; others, however, may need to be performed on all copies, as they are related to the common functionality. Cloning therefore creates the risk of changes not being propagated to all copies. While not all code copying is detrimental to software quality, copying is far too common in practice. Many software errors are due to diverging clones, and thus it is important to be able to find such clones in order to investigate whether changes in one should also have been made to others; better yet, clones should be detected when checking-in code to the code repository, and the developers alerted immediately.

In many cases, abstracting common behavior to reduce clones can improve code quality. Another motivation for our work on clone detection is the availability of our Extract Computation refactoring tool [4], which can use information

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 502–526, 2013.

from the comparison of found clones to automatically extract common parts (see Section 6 for a further discussion of this issue).

Sometimes code is copied illegally; this can be intentional plagiarism, but, given the complexity of software licenses, can also happen as a result of a mistaken belief by a developer that a certain piece of code can legally be copied. Software-development companies need to protect themselves against both kinds of violations.

Clone detection tools [5,6,7,8,9,10,11,12] attempt to find copied code, to assist in all challenges listed above. They use a variety of techniques in order to identify clones that may have been changed in various ways. Roy et al. [13] classify code clones into four types, depending on the variations allowed between the copies. Type-1 clones vary in whitespace, layout, and comments; type-2 clones can additionally vary in identifiers, literals, and types; type-3 clones may also have changed, added, or removed statements; and type-4 clones are broadly defined as "code fragments that perform the same computation but are implemented by different syntactic variants." Clone detection tools use various techniques, and address different parts of this space. Syntax-oriented tools typically handle type-2 and some type-3 clones (depending on the amount of code inserted, changed, or removed). Tools that employ more semantic representations can also find some type-4 clones.

The availability of automated refactoring support in modern development environments (such as Eclipse,[1] IntelliJ,[2] and Microsoft Visual Studio[3]) further complicates the task of clone detection, as they make it very easy for developers (and plagiarists) to make significant and wide-ranging syntactic changes to code without changing its functionality. For example, the task of manually renaming a variable, field, or method is tedious and error-prone; because the same name is often used in multiple ways, it is easy to change the wrong one or forget to change the right one. With a refactoring tool, this task is almost effortless, leading to the proliferation of type-2 clones.

Refactorings such as Extract Method (and its inverse, Inline Method) typically create type-4 clones, by replacing a potentially large piece of code with a method call. This results in significant changes to the program syntax, and also to the graph representations used by semantic clone detection tools; this makes it much more difficult for these tools to find such clones.

We consider the detection of refactored clones to be an important research problem, with implications both for code quality and plagiarism discovery. As part of this research, we developed an algorithm and a tool, called Cider,[4] to find clones despite refactorings such as Rename, Extract Local Variable, Inline Variable, and, most significantly, Extract Method and Inline Method.

Soares et al. [14] analyzed the changes in large open-source projects. They concluded that 27% of the changes in those project can be attributed to refac-

[1] `http://www.eclipse.org`
[2] `http://www.jetbrains.com/idea`
[3] `http://msdn.microsoft.com/vstudio`
[4] Standing for "Clone Identifier."

toring and out of those more than third (36%) were inter-procedural; our results (see Section 4) are consistent with this finding.

Section 2 presents a motivating example, taken from one of the open-source projects we used to evaluate Cider. Section 3 presents the algorithm, with an evaluation in Section 4. Section 5 summarizes related work, and Section 6 concludes with a discussion and future work.

2 Example

The code in Figure 1 is taken from version 4.9 of the JUnit project.[5] The class `FailOnTimeout` consists of a single constructor and a single method. In version 4.10, this class was refactored to the form shown in Figure 2, which still has a single constructor but now contains multiple methods, as well as an inner class (shown in Figure 3). Superficial inspection is likely to suggest that these two classes are quite different, although these are in fact type-4 clones.

The code of version 4.9 was refactored in several steps to arrive at that of version 4.10. The most significant change is the creation of the class `StatementThread` (Figure 3) to encapsulate information about the thread used in the `evaluate` method of Figure 1. This information includes the field `fFinished` of the original class, as well as `fThrown`, which has been renamed to `fExceptionThrownByOriginalStatement`. The field `fTimeout` has remained unchanged, while `fNext` has been renamed to `fOriginalStatement`.

The constructors of class `FailOnTimeout` in the two versions are identical up to renaming (i.e., are type-2 clones). However, the `evaluate` method has changed in several ways. The thread in version 4.9 had the static type `Thread`, and the object's actual type is that of the anonymous inner class defined in the `evaluate` method; in version 4.10 it has the type `StatementThread` which inherits from `Thread`.

The value of the `Statement` object in the enclosing class is copied to the field `fStatement` of the inner class; but it is used in the `run` method of that class in the same way that the original field was used in the corresponding method of the anonymous inner class in the `evaluate` method of version 4.9. The only piece of brand new code in the `run` method is the clause that catches exceptions of type `InterruptedException`.

After the thread is created, it is run and joined in exactly the same way; this computation happens inside the `evaluate` method of Figure 1, but in Figure 2 these operations have been extracted into the new `evaluateStatement` method. The latter, however, has another brand new statement, which interrupts the thread. The computation now continues in the `evaluate` method of both versions, but the test of the field `fFinished` has been replaced by testing the corresponding field of the `StatementThread` object, and the sense of the comparison has been reversed without changing the behavior.

[5] `http://www.junit.org`

```
public class FailOnTimeout extends Statement {
  private Statement fNext;
  private final long fTimeout;
  private boolean fFinished= false;
  private Throwable fThrown= null;

  public FailOnTimeout(Statement next, long timeout) {
    fNext= next;
    fTimeout= timeout;
  }

  @Override
  public void evaluate() throws Throwable {
    Thread thread= new Thread() {
      @Override
      public void run() {
        try {
          fNext.evaluate();
          fFinished= true;
        } catch (Throwable e) {
          fThrown= e;
        }
      }
    };
    thread.start();
    thread.join(fTimeout);
    if (fFinished)
      return;
    if (fThrown != null)
      throw fThrown;
    Exception exception= new Exception(String.format(
      "test timed out after %d milliseconds", fTimeout));
    exception.setStackTrace(thread.getStackTrace());
    throw exception;
  }
}
```

Fig. 1. Example code from the JUnit project, version 4.9

```
public class FailOnTimeout extends Statement {
  private final Statement fOriginalStatement;
  private final long fTimeout;

  public FailOnTimeout(Statement originalStatement, long timeout) {
    fOriginalStatement= originalStatement;
    fTimeout= timeout;
  }

  @Override
  public void evaluate() throws Throwable {
    StatementThread thread= evaluateStatement();
    if (!thread.fFinished)
      throwExceptionForUnfinishedThread(thread);
  }

  private StatementThread evaluateStatement()
      throws InterruptedException {
    StatementThread thread= new StatementThread(fOriginalStatement);
    thread.start();
    thread.join(fTimeout);
    thread.interrupt();
    return thread;
  }

  private void throwExceptionForUnfinishedThread(StatementThread thread)
      throws Throwable {
    if (thread.fExceptionThrownByOriginalStatement != null)
      throw thread.fExceptionThrownByOriginalStatement;
    else
      throwTimeoutException(thread);
  }

  private void throwTimeoutException(StatementThread thread)
      throws Exception {
    Exception exception= new Exception(String.format(
      "test timed out after %d milliseconds", fTimeout));
    exception.setStackTrace(thread.getStackTrace());
    throw exception;
  }

  private static class StatementThread extends Thread {
    // See Figure 3
  }
}
```

Fig. 2. Code from Figure 1, as refactored in version 4.10 of the JUnit project

```
private static class StatementThread extends Thread {
  private final Statement fStatement;
  private boolean fFinished= false;
  private Throwable fExceptionThrownByOriginalStatement= null;

  public StatementThread(Statement statement) {
    fStatement= statement;
  }

  @Override
  public void run() {
    try {
      fStatement.evaluate();
      fFinished= true;
    } catch (InterruptedException e) {
    } catch (Throwable e) {
      fExceptionThrownByOriginalStatement= e;
    }
  }
}
```

Fig. 3. Inner class from Figure 2

The next part of the original code, which throws either the exception caught by the thread's `run` method if there was one, or a new timeout exception, has been extracted into two new methods. Checking which exception to throw is done in `throwExceptionForUnfinishedThread`, which checks a field of the inner class instead of a field of the outer class. The creation and throwing of the timeout exception has been extracted into the new method `throwTimeoutException`.

This analysis demonstrates that the two versions perform identical computations, with only one real addition (interrupting the thread). However, the two versions are quite different syntactically, and state-of-the art clone-detection tools do not do well with this code. Deckard [7] was able to locate very few similar code fragments, mostly in the non-functional part of the files (the import section). CCFinderX [8] performed better, but reported six different clone sets, each containing several clone regions. Neither tool reported the `evaluate` method of version 4.10 (Figure 2) as being part of a clone. This outcome is not surprising, as both tools rely on the syntactic properties of the source code, which have changed significantly between the two version. (Both Deckard and CCfinderX were used with various settings; the best results were obtained using non-default setting to locate relatively small clones.) Sematic clone detectors such as Komondoor and Horwitz's [10], Duplix [11], GPLAG [15], and Gabel et al.'s [12] are not expected to perform much better, as the clone analysis applied by these tools is limited to the boundaries of a single method. Hence, these tools will also report multiple small clones, if any. Unfortunately, these tools were not available for us to experiment with.

3 The Cider Algorithm

Cider is a new clone-detection tool specifically aimed at finding clones that have diverged by one or more refactoring steps. This section explains the underlying representation, how the matching algorithm works, and how initial seeds are found (in two different scenarios), and demonstrates how the algorithm works on the example of Section 2.

3.1 The Internal Representation

Like other graph-based algorithms, Cider searches for clones on a graph representation of the program, starting from a set of initial seeds. Unlike other algorithms, however, Cider uses the plan calculus [16,17] to represent programs. The plan representation is canonical to a large extent, in the sense that different syntactic variants expressing the same computation are represented by the same plan. For example, because data flow is represented directly by edges rather than through the use of variable names, the representation is insensitive to names of local variables (although this information is available if needed). The plan representation has been used in several projects, including some that were commercially successful [18,19].

Figure 4 shows the plan representation of the `evaluate` method of Figure 2, as generated (and drawn) by Cider. The plan is a graph whose nodes are computational elements (drawn as rectangles), tests that split control flow into two parts (drawn as split boxes with two bottom sections, corresponding to the true and false branches), and joins that serve the dual purpose of merging control flows and at the same time indicating which data flows from which incoming side (drawn as split boxes with two sections on top, representing the control flows being joined). The plan has two types of edges, indicating data flow (full edges, labeled with the variables or temporary values they carry) and control flow (dashed edges). Side effects on objects are represented by data flow that carries the modified object; Figure 4 shows the current object `this` as an input as well as an output of this method, as well as of called methods that might potentially modify it.

It is possible to generate the plan representation at various levels of granularity. Each statement could be represented by a single operation node, as is typical of other representations such as the Program-Dependence Graph (PDG) [20]. This makes the representation insensitive to changes inside individual statements, as long as the data and control flow between statements remain the same. For example, inserting a negation in an expression would not change the graph of such a coarse-grained plan (nor the PDG representation). Our current implementation of the plan formalism uses a finer-grained approach, in which each operation is represented as a separate node. This means that modifying expressions does change the structure of the graph, requiring more sophisticated matching (see Section 6). On the other hand, statement boundaries no longer affect the graph; for example, the Extract Local Variable refactoring, which replaces an expression by a variable and adds an assignment statement preceding

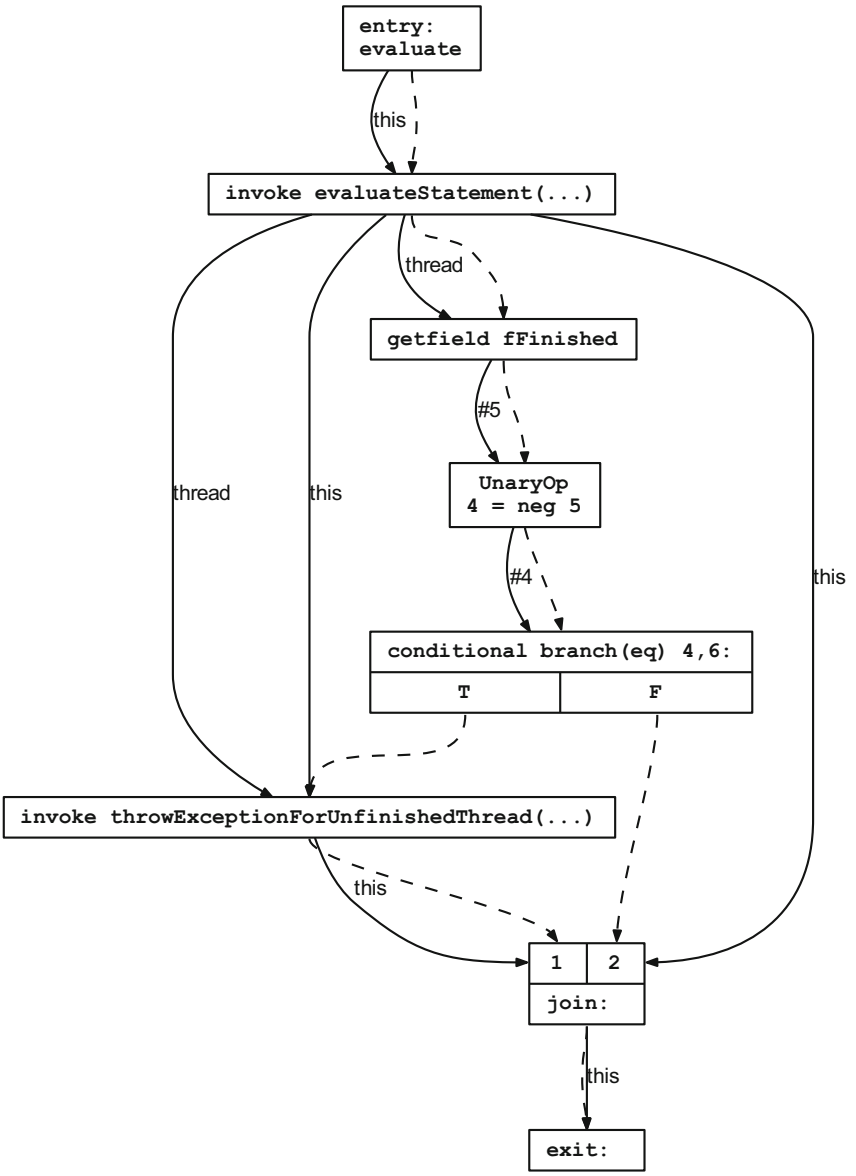


Fig. 4. The plan representation of the `evaluate` method of Figure 2

the statement that contained the expression, does not change the structure of the graph. Since one of our goals is to use our Extract Computation refactoring [4] to automatically re-combine the common parts of detected clones, sensitivity to expressions and not to statements is appropriate.

3.2 Finding Interprocedural Clones

The Cider algorithm is similar in general form to Krinke's [11], but extended for interprocedural matching. The algorithm starts with a seed, which is a proposed match between two plan nodes. The algorithm attempts to enlarge the seed to a match between two sub-plans, by successively adding new nodes to the match, searching both forwards and backwards, in order to obtain as large a match as possible. Unlike other graph-matching algorithms, Cider searches for new nodes by matching paths rather than single edges. This is necessary for several reasons, most (but not all) of which have to do with the fact that the search is interprocedural.

In the case of control-flow edges, the algorithm must extend paths that connect to test or join nodes, since the clone may match only one side of a conditional; for example, the fragment

```
A; if (p) B else C; D;
```

should match the fragments[6]

```
A; B; D;
```
and
```
A; C; D;.
```

Similarly,

```
A; f(); C;
```

where the body of `f` is `B` should match

```
A; B; C;.
```

This requires a single control flow edge in the latter fragment to match a path that goes from `A` through the call to `f` to `B`, and likewise for the path from `B` through the exit node of `f` to `C`.

Suppose now that the method `f` has a parameter whose value is computed in `A` and used in `B`, and that `f` returns a value used in `C`. The data-flow paths from the source of the parameter to its use in `B`, and from the source of the return value to its use in `C`, each corresponds to a single edge in `A; B; C;`. The Cider algorithm recognizes these cases and follows paths through tests, joins, method calls, and returns, in order to allow these kinds of matches. Because Cider is built on top of WALA,[7] it can utilize WALA's pointer analysis to provide information about possible targets for polymorphic calls, improving the algorithm's accuracy.

When matching nodes, Cider uses a configurable similarity test, to support various levels of strictness in the search. Each node in the plan has an associated semantics; in the case of the primitive language operators (including object construction with `new`), it indicates which operator it is; for method calls, it indicates

[6] In our current implementation, this will only work if there is data flow from `A` to `B` (or `C`), or if `p` is a variable whose computation precedes `A`, since otherwise the computation of `p` intervenes between `A` and the test node. See Section 6 for a discussion of ways to overcome this limitation.

[7] The T. J. Watson Libraries for Analysis, `http://wala.sourceforge.net`

which method is called. In the prototype described in this paper, nodes match only if they have the same semantics; names, however, are not required to match (see Algorithm 5). This makes the match insensitive to method names, and may erroneously match unrelated methods that happen to have similar implementations. It is possible to require names to match perfectly; this will make the match sensitive to method names (unlike names of local variables), and will not find clones that underwent renaming of methods (or classes). A middle approach is also possible; for example, by using tools such as RefactoringCrawler [21] that attempt to discover renamings and allow those to match. Cider does not require both data flow and control flow to match for the same node. This allows some variability in the clones, since code modifications that only affects control flow will be ignored by the search through data-flow edges, and vice versa.

More formally, the matching algorithm receives two plans to be matched, a seed, consisting of an initial match between two semantically-equivalent nodes, one from each plan, and a scope that limits the search.[8] Plan nodes are classified as either *computational* or *intermediary*, based on their type and the given scope. Computational nodes must match between the clones, but intermediary nodes are ignored. Operation nodes are computational, and all tests and joins are intermediary. Method call nodes and method entry and exit nodes are considered intermediary if they are in scope and their source code is available, and computational otherwise.

We define a *flow* to be a path in the interprocedural plan that starts and ends in computational nodes, and all whose internal nodes (if any) are intermediary. All edges in a single flow have the same type (data or control). The algorithm matches a flow in one plan to another flow in the other. A *clone candidate* is a collection of plan nodes and flows connecting them. The algorithm is shown as Algorithm 1.

In each iteration, the algorithm computes a frontier for each clone candidate. The frontiers, computed by Algorithm 2, consist of all flows that bridge the candidate with other code. However, a history is kept so that each edge appears in the frontier only once. This is done in order to prevent an edge being matched with an irrelevant part of the second candidate. (The next version of the algorithm will support flow that cross computational nodes, in order to allow the addition or removal of arbitrary code (see Section 6). In that version, the algorithm keeps a mapping between matched nodes in the two candidates, and the history is not necessary.) Frontiers from one candidate are compared to those of the other, using a flexible similarity criterion expressed by the function called "similar"; the version we used in the evaluation is shown in Algorithm 5. Finally, matching flows are added to both candidates, using the expand function (Algorithm 4), and the process repeats.

[8] For simplicity, we describe the algorithm as working on two clones candidates, although it is able to handle more than two simultaneous candidate clones. Similarly, the seeds can be generalized from single nodes to be sub-graphs that have already been matched in some other way.

Algorithm 1. find-clone-candidates((s_1, s_2): seed, p_1: plan, p_2: plan, c: scope) returns a pair of clone candidates

{Initialize clone candidates}
$\text{cand}_1 \leftarrow \{s_1\}$
$\text{cand}_2 \leftarrow \{s_2\}$
{Initialize frontier histories}
$H_1 \leftarrow \emptyset$
$H_2 \leftarrow \emptyset$
repeat
  $F_1 \leftarrow \text{frontier}(\text{cand}_1, p_1, c, H_1)$
  $F_2 \leftarrow \text{frontier}(\text{cand}_2, p_2, c, H_2)$
  $\text{matches} \leftarrow \{(f_1 \in F_1, f_2 \in F_2) \mid \text{similar}(f_1, f_2)\}$
  $\text{expand}(\text{cand}_1, \{f_1 \mid \exists f_2.\ (f_1, f_2) \in \text{matches}\})$
  $\text{expand}(\text{cand}_2, \{f_2 \mid \exists f_1.\ (f_1, f_2) \in \text{matches}\})$
until $\text{matches} = \emptyset$
return $(\text{cand}_1, \text{cand}_2)$

Algorithm 2. frontier(cand: clone-candidate, p: plan, c: scope, H: history) returns a set of flows; modifies H

$E_{\text{forward}} \leftarrow \{e \in p \mid \text{source}(e) \in \text{cand} \wedge \text{target}(e) \notin \text{cand}\}$
$E_{\text{backward}} \leftarrow \{e \in p \mid \text{source}(e) \notin \text{cand} \wedge \text{target}(e) \in \text{cand}\}$
$F \leftarrow \bigcup_{e \in E_{\text{forward}}} \text{flow-forward}(e, p, c) \cup \bigcup_{e \in E_{\text{backward}}} \text{flow-backward}(e, p, c) \setminus H$
$H \leftarrow H \cup F$
return F

Algorithm 3. flow-forward(f: edge or flow, cand: clone-candidate, c: scope, p: plan)
returns a set of flows

if target(f) is computational according to c **then**
  return $\{f\}$
else
  $\text{next} \leftarrow \text{outgoing-edges}(\text{type}(f), \text{target}(f), p)$
  $\text{next} \leftarrow \text{filter}(\text{next}, c)$
  if $\text{next} = \emptyset$ **then**
    return $\emptyset$
  else
    return $\bigcup_{e \in \text{next}} \text{flow-forward}(f \,||\, e, p, c)$
  end if
end if

Frontiers are computed starting with edges that exit the candidate, both forwards and backwards, extending them to complete flows by the flow-forward and flow-backward functions. Both flow functions work on partial flows, which may end in intermediary nodes; however, each returns a set of full flows. The flows in the frontiers of both candidates are then compared for similarity; those that

Algorithm 4. expand(cand: clone-candidate, F: set of flows)
modifies cand

```
for all f ∈ F do
  for all node n in f do
    add n to cand
  end for
end for
```

Algorithm 5. similar(f_1: flow, f_2: flow)

return type(f_1) = type(f_2) ∧
semantics(source(f_1)) = semantics(source(f_2)) ∧
semantics(target(f_1)) = semantics(target(f_2))

are found to be similar are used to extend the two candidates. The process ends when the candidates cannot be extended any further, and the final candidates are returned. At that point, filters can be applied to decide whether the returned candidates should really be considered to be clones, based, for example, on their sizes.

The flow-forward function (Algorithm 3) computes flows for edges whose source is in the candidate but whose target is outside it. (The flow-backward function, not shown here, is analogous.) It tries to extend incomplete flows (i.e., those that end in an intermediary node) by considering all outgoing edges from the last node of the flow; these can be data or control edges, depending on the type of the flow. This set is pruned by the filter function, based on the scope. This leaves room for various criteria to be applied at this stage.

3.3 Finding Seeds

We consider two different types of scenarios for Cider. In the first, we are looking for a relatively small number of clones in a large corpus (or between two corpora, in the case of a search for plagiarism). In the second, we are looking for non-identical clones between two versions of the same software system; in this case, most of the code is identical, and we are looking for changes that look different but have very similar functionality, as in the example of Section 2. While the matching algorithm described in Section 3.2 is the same in these two scenarios, the computation of the seeds is necessarily different.

In the first scenario there are many potential seeds. For example, any addition operation in the program may be matched to any other such operation. The main challenge is to find good seeds that will lead to clones that are large enough to be interesting. The Cider algorithm is relatively expensive, and it is impossible to apply it to every such pair. We therefore use a hybrid approach, where a more efficient clone detector is used to winnow the potential seeds based on (mostly) syntactic criteria. For example, we have used Deckard [7] to provide initial seeds for Cider. We used Deckard with somewhat unusual search parameters, which are more suitable for finding seeds for Cider than for finding actual

clones. Specifically, we used strong criteria for matches, but accepted clones that are relatively small and possibly insignificant by themselves.

The second scenario requires a different strategy. Here, almost all the code is identical and would normally be considered to be cloned. We are interested in code that changed in certain ways; specifically, by method extraction or inlining. We call such cases *interprocedural clones*, or *IPC* for short. We therefore look for method calls that appear in only one version but not the other. This is done by comparing the call graphs of the two versions, looking for a method in one version containing a call that does not appear in the corresponding place in the other version. We call these *IPC candidates*. Our current implementation searches for methods with identical signatures (including names), with some call that appears only in one version. The starting criterion can obviously be extended, for example, by using renamed candidates suggested by RefactoringCrawler [21] or similar tools.

Each IPC candidate is then analyzed to see whether it really is an interprocedural clone, some other kind of clone, or not a clone at all. In order to do that, we need to find seeds in the corresponding methods. There can still be a prohibitively large number of such seeds, and so an additional heuristic is needed in order to prune their number. We start by selecting pairs of identical operations (according to the similarity function mentioned above). For each pair we apply the Cider algorithm with a scope that does not cross method boundaries (making it intraprocedural). Each pair results in two matching sub-plans, which may be small. We select as seeds only those pairs of operations that appear in a large fraction of the generated sub-plans; these are likely to be central operations in the clones, and are more likely to yield interprocedural clones with the full version of the Cider algorithm. By experimenting on one set of IPC candidates, we came up with the following step threshold function. If there are up to 10 pairs, we require seeds to appear in at least 50% of the sub-plans; if there are between 11 and 50 initial pairs, we require seeds to appear in 80% of the sub-plans; between 51 and 100 the requirement drops to 70%, and then to 60%. The rationale for this choice is that as the number of initial pairs grow, we are more likely to discover separate clone "islands," and requiring too high a percentage will eliminate most of the matches that could lead to interprocedural clones.

3.4 Cider Example

This section shows an example of how the Cider algorithm can be applied to the JUnit example of Section 2. In this experiment, we manually chose the two operations that access the `fTimeout` field (in the `evaluate` method of Figure 1 and the `evaluateStatement` method of Figure 2) as a seed, and set the scope of the exploration to include all the methods in the class.

Starting from the chosen seed, the algorithm proceeds to compute the frontiers (Algorithm 2). The backward frontier in version 4.9 is:

Control: `[invoke start(...)]` → `[getfield fTimeout]`
Data: `[entry: evaluate]` → `[getfield fTimeout]`;

the data edge is due to the data flow of the `this` pointer from the entry of the method to the `getfield` operation.

The forward frontier is:

Control: `[getfield fTimeout]` → `[invoke join(...)]`
Data: `[getfield fTimeout]` → `[invoke join(...)]`.

The data edge is due to the result of the `getfield` operation, which is used as the parameter to the `thread.join()` call. The call to `thread.join()` is not considered an intermediary node, since this is a library method and is outside the search scope. Similarly, the entry of the `evaluate` method is out of scope, since no callers of this method exist inside the class definition.

In version 4.10, the control predecessor of the `getfield` operation is the preceding node, `invoke start`, as in version 4.9. The data predecessor, in contrast, is the entry node of `evaluateStatement`, since the data for the field `fTimeout` comes from the current object (`this`), which is unchanged from its value on entry. This entry node, however, is an intermediary node, and so the `flow-backward` function will extend this edge to a longer flow, through the call of `evaluateStatement` in `evaluate` to the entry of `evaluate`; this is a computational node, since no callers are in scope. The backward frontier in version 4.10 is therefore:

Control: `[invoke start(...)]` → `[getfield fTimeout]`
Data: `[entry: evaluate(...)]` → `[getfield fTimeout]`;

The forward frontier is same as in the other version:

Control: `[getfield fTimeout]` → `[invoke join(...)]`
Data: `[getfield fTimeout]` → `[invoke join(...)]`

The two `[entry: evaluate]` nodes obviously match, as do the `[invoke start(...)]` and `[invoke join(...)]` nodes, and all three pairs are added to the clone candidates.

Following the data flow of the `thread` pointer forward from the `thread.join()` call in version 4.9 leads to `thread.getStackTrace()` (just before the exception is thrown on the last line). Following both control and data flow from `thread.join()` in version 4.10 lead to the following statement, `thread.interrupt()`. Because the similarity function ignores names, these are considered to match, and both are added to the clone candidates. In the next step, data flow from `thread.getStackTrace()` in version 4.9 leads to `exception.setStackTrace()` on the same line. In version 4.10, tracing data flow from the `thread.interrupt()` call leads, through the exit of `evaluateStatement` back to the body of `evaluate`, then into the call to `throwExceptionForUnfinishedThread`, to the `throwTimeoutException()` call, and finally to `thread.getStackTrace()` call in the next-to-last line of that method. Again, because similarity ignores names, this pair is added to the clone candidates.

Following flows from the matching entries of `evaluate()` will produce some additional matches, resulting in the clones identified in Figures 5 and 6. In both cases, lines in the detected clones are marked with two plus signs. This result

```
  public class FailOnTimeout extends Statement {
    private Statement fNext;
    private final long fTimeout;
    private boolean fFinished= false;
    private Throwable fThrown= null;

    public FailOnTimeout(Statement next, long timeout) {
      fNext= next;
      fTimeout= timeout;
    }

    @Override
++  public void evaluate() throws Throwable {
++    Thread thread= new Thread() {
        @Override
        public void run() {
          try {
            fNext.evaluate();
            fFinished= true;
          } catch (Throwable e) {
            fThrown= e;
          }
        }
      };
++    thread.start();
++    thread.join(fTimeout);
      if (fFinished)
        return;
      if (fThrown != null)
        throw fThrown;
++    Exception exception= new Exception(String.format(
++      "test timed out after %d milliseconds", fTimeout));
++    exception.setStackTrace(thread.getStackTrace());
++    throw exception;
++  }
  }
```

Fig. 5. Clone detected in Figure 1; compare with Figure 6

is very close to the best that a human would find. These clones do not include the code in the `run()` methods of the two `Thread` subclasses, since these are not reachable from the found clones; they can be discovered by an different seed.

The second clone erroneously includes the `thread.interrupt()` call of version 4.10, due to the wrong matching of this call with `thread.getStackTrace()` in version 4.9. Such inaccuracies are balanced by a careful choice of seeds, as described in Section 3.3. Section 6 describes the changes we are making in our next prototype, which will be sensitive to names, but will explore flows that

```
  public class FailOnTimeout extends Statement {
    private final Statement fOriginalStatement;
    private final long fTimeout;

    public FailOnTimeout(Statement originalStatement, long timeout) {
      fOriginalStatement= originalStatement;
      fTimeout= timeout;
    }

    @Override
++  public void evaluate() throws Throwable {
++    StatementThread thread= evaluateStatement();
      if (!thread.fFinished)
++      throwExceptionForUnfinishedThread(thread);
++  }

++  private StatementThread evaluateStatement()
        throws InterruptedException {
++    StatementThread thread= new StatementThread(fOriginalStatement);
++    thread.start();
++    thread.join(fTimeout);
++    thread.interrupt();
++    return thread;
++  }

++  private void throwExceptionForUnfinishedThread(StatementThread thread)
        throws Throwable {
      if (thread.fExceptionThrownByOriginalStatement != null)
        throw thread.fExceptionThrownByOriginalStatement;
      else
++      throwTimeoutException(thread);
    }

++  private void throwTimeoutException(StatementThread thread)
        throws Exception {
++    Exception exception= new Exception(String.format(
++      "test timed out after %d milliseconds", fTimeout));
++    exception.setStackTrace(thread.getStackTrace());
++    throw exception;
++  }
  }
```

Fig. 6. Clone detected in Figure 2; compare with Figure 5

bypass some nodes in order to find matches in spite of added statements. We expect that version to find more accurate clones in this example, including the identification of the correspondence between clone nodes.

Table 1. Benchmark project statistics

Project	Version	SLOC	Methods	IPC Candidates	Cider Run Time (sec)
JUnit	4.9	15629	2441	7	2.4
	4.10	16434	2539		
HSQLDB	2.2.7	158972	10129	33	24.5
	2.2.8	159278	10166		
PDFBox	1.5	52791	4388	135	318.8
	1.6	55316	4531		
PMD	4.3	62369	6072	122	74.2
	5.0.1	75959	7560		

4 Evaluation

Because Cider's major difference from other clone-detection tools is its ability to find clones that have diverged by refactoring, we wanted the evaluation to concentrate on such clones, and, in particular, on clones that underwent method extraction. We therefore chose to search for such clones between successive versions of the same software project. We used a simple threshold seed oracle as described in Section 3.3, and limited the scope of the search to the method containing the candidate.[9]

Table 1 summarizes the relevant characteristics of the open-source projects on which we tested the tool. For each project, we chose two versions on which to perform the evaluation. JUnit (from which we took the example of Section 2) is a unit testing framework for Java; HSQLDB[10] is a relational database engine; PDFBox[11] is a library for working with PDF documents; and PMD[12] is a Java code analyzer. These projects range in size from JUnit's 16K SLOC to HSQLDB's 160K SLOC,[13] and contain from 2500 methods (JUnit) to over 10000 (HSQLDB). The last column in the table displays the number of IPC candidates discovered by the call graph analysis described in Section 3.3. This number is reasonably small, enabling our untuned implementation to check all candidates within seconds or minutes (the precise times are given in the last column of Table 1; each number is Cider's running time on the given sample in seconds, as an average of 10 runs).

Cider classified each IPC candidate into one of three categories: (1) true interprocedural clones (IPC); (2) clones that are not interprocedural (NIPC); or (3) not clones (NC). Out of all 291 IPC candidates that were found in the sam-

[9] Note that this scope is smaller than the one used in the example of Section 3.4.
[10] `http://hsqldb.org`
[11] `http://pdfbox.apache.org`
[12] `http://pmd.sourceforge.net`
[13] As determined by CLOC (`http://cloc.sourceforge.net`)

Table 2. Recall and precision in the identification of interprocedural clones (IPC) in IPC candidates

Project	Manual	Cider	Both	Recall(%)	Precision(%)
JUnit	4	3	3	75	100
HSQLDB	4	4	4	100	100
PDFBox	7	14	6	86	43
PMD	13	13	10	77	77
Total	28	34	23	82	68

Table 3. Recall and precision in the identification of clones (IPC and NIPC) in IPC candidates

Project	Manual	Cider	Both	Recall(%)	Precision(%)
JUnit	7	7	6	86	86
HSQLDB	10	10	10	100	100
PDFBox	43	40	32	74	80
PMD	33	33	28	85	85
Total	93	90	76	82	84

ple projects, we used 43 as a training set to adjust the threshold function. In order to assess the accuracy of Cider's results, we manually classified a random sample consisting of 117 of the remaining 248 IPC candidates and compared our manual classification with that of Cider. We computed recall and precision[14] for the decision of whether the candidate is an interprocedural clone (i.e., classified as IPC), and whether it is a clone at all (i.e., IPC or NIPC). Table 2 shows the results of the first comparison, and Table 3 shows the results of the second. In each table, the column headed "Manual" gives the number of cases in which we decided that the criterion holds based on manual inspection; the "Cider" column gives the number of cases identified by Cider, and the "Both" column gives the number of cases in which both agreed that the criterion holds (i.e., the number of true positives).

First, it is interesting to note that there were interprocedural clones in all projects; in total, 24% of the IPC candidates in the sample were found to be real interprocedural clones by manual inspection. Cider performs quite well both in terms of precision and recall. We investigated several of the cases in which Cider's classification disagreed with ours. In some cases, we found these to be artifacts of the way some language features are implemented.

For example, Figure 7 shows two versions of the `compareTo` method of class `COSWriterXRefEntry` in PDFBox. The class has been refactored to use generics,

[14] For a given criterion, recall is the percentage of cases that the algorithm correctly classified out of those that actually match the criterion, and indicates the frequency of false negatives. Precision is the percentage of cases that the algorithm correctly classified out of those it classified as matching the criterion, and indicates the frequency of false positives. In both cases, 100% indicates the best recall or precision.

```
public class COSWriterXRefEntry implements Comparable {
  // ...
  public int compareTo(Object obj)
  {
    if (obj instanceof COSWriterXRefEntry)
    {
      return (int)(getKey().getNumber()
        - ((COSWriterXRefEntry)obj).getKey().getNumber());
    }
    else
    {
      return -1;
    }
  }
  // ...
}
```

(a)

```
public class COSWriterXRefEntry implements Comparable<COSWriterXRefEntry>
  // ...
  public int compareTo(COSWriterXRefEntry obj)
  {
    if (obj instanceof COSWriterXRefEntry)
    {
      return (int)(getKey().getNumber() - obj.getKey().getNumber());
    }
    else
    {
      return -1;
    }
  }
  // ...
}
```

(b)

Fig. 7. The `COSWriterXRefEntry.compareTo` method in PDFBox: (a) version 1.5; (b) version 1.6

and so the method signature changed from `compareTo(Object)` to `compareTo (COSWriterXRefEntry)`. (In fact, this change has been done carelessly; the conditional is no longer necessary in the generic version.) There are no other changes, and from a developer's point of view there is no interprocedural clone. However, the method `compareTo(Object)` still exists in the new class, even though it has no representation in the text; this method checks the dynamic type of its argument, and forwards to the type-specific method as appropriate. It is the two `compareTo(Object)` methods that Cider compares, and it correctly identifies an interprocedural clone.

In other cases, Cider found relatively small sub-plans it considered spuriously to be interprocedural clones. We intend to experiment with the thresholds on the sizes of the matching parts in the calling and called methods to see how they affect recall and precision.

5 Related Work

Komondoor and Horwitz [10] proposed the use of the PDG for clone detection. By employing a lock-step slicing [22] technique using a combination of backward slicing with limited forward slicing, they were able to compute isomorphic subgraphs of the PDG. They seeded their algorithm with a set of syntactically-equivalent node pairs, and performed backward slicing from each pair with a single forward slicing step for matching predicates nodes. Their method was applied to several open-source software systems written in the C programming language, with results demonstrating the capability of slicing to detect non-contiguous code clones.

Krinke [11] introduced Duplix, which uses a fine-grained PDG to represent procedures. Clones are identified by maximal similar subgraphs, which are constructed in a lock-step manner starting from a pair of predicate nodes and extending the subgraph by adding similar outgoing edges. This process is more general than that used by Komondoor and Horwitz; Cider's graph search is similar to Krinke's.

GPLAG [15] uses a relaxed form of subgraph isomorphism of PDGs for the purpose of plagiarism detection. The authors note that PDGs are resilient to some semantics-preserving modifications, like (unrelated) statement insertion, statement reordering, and control replacement. Before attempting to locate isomorphic subgraphs, GPLAG reduces the search space by filtering methods based on size and structural similarity metrics. This heuristic is likely to fail when Extract or Inline Method have been performed.

Jiang et al. [7] focused their attention on the scalability of semantic clone detection. As a first step, they introduced Deckard, a token-based detection tool that maps trees to feature vectors, and uses distance metrics to find similar trees. Subsequently, Gabel et al. [12] extended this approach to handle subgraph similarity by first reducing carefully selected PDG subgraphs to their related abstract syntax trees, then applying Deckard to those trees. They report that this approach can scale to millions of lines of code.

The above-mentioned tools do not attempt to find interprocedural clones. Godfrey and Zou [23] used what they call "origin analysis" to detect splits and merges in procedural source code. (These terms correspond to the effects of the Extract Method and Inline Method refactorings, respectively.) Origin analysis consists of entity and relationship analysis. Each entity (function) in the code is represented by a set of metrics, such as number of lines of code, number of local/global variables used, cyclomatic complexity, etc. Relationship analysis is computed using call graph analysis to compute caller/callee relationships. The authors implemented their technique as part of the semi-automatic Beagle tool,

whose goal is to help developers gain insight of a system's evolution. To detect splits and merges, the user specifies an entity of interest and a set of entities the tool should analyze.

While their tool is able to detect simple split and merge transformations based on syntactic metrics, it is susceptible to other kinds of changes that may alter the metrics of the extracted code. These could include inlining or extracting a local variable, which changes the number of lines of code; field encapsulation, which creates new caller/callee relations where none existed before; and other simple refactorings that Cider is oblivious to.

Like Godfrey and Zou, Dig et al. [21] attempt to infer refactoring transformations between two version of a project. First, they find matching code entity pairs, where an entity can be a method, class, or package, using shingles (a metric-based fingerprinting method). Then they employ a more expensive semantic analysis to determine the relationships between entities. They use a set of strategies in a fixed order that enables them to take advantage of previously-inferred refactorings; for example, information about package renaming can be used when searching for method renaming. Their focus is on API-level refactoring such as Rename, Move, Pull Up Method, and Push Down Method rather than code-level refactorings such as Extract Method or Inline Method.

6 Discussion and Future Work

The fact that we found interprocedural clones in every project that was evaluated strongly suggests that such clones are ubiquitous. The occurrence of interprocedural clones is likely to grow further as refactoring tools improve, and developers' awareness of such tools increases. The same is true for other refactorings, such as Rename, Extract Local Variable, etc. Furthermore, modern development environments provide other types of automatic code transformations, such as adding or removing blocks around single statements, or inverting a conditional and switching the "then" and "else" parts. (In Eclipse, these are provided as part of the Quick Fix facility.)

These tools are very convenient for the programmer (or plagiarist), but create a challenge for clone-detection tools that need to discover the equivalence of the code after one or more such changes have been applied to the original version. The changes can be quite extensive, even crossing module boundaries. For example, it is easy to follow an Extract Method refactoring by Move Method, Pull Up, or Push Down, causing part of the code clone to move to another source file.

As mentioned at the end of Section 2, syntax-based clone detectors cannot cope with such changes. They can only discover matches between code segments that remained together, so that they will find smaller fragments, which may be below their reporting thresholds. Even if they are larger, discovering interprocedural clones would still require manual inspection. The semantic clone detectors we are aware of limit themselves to a single method, and do not find interprocedural clones either.

In order to discover clones despite extensive refactoring, new, more semantic, techniques are required. Cider demonstrates several such techniques, from the use of data edges instead of relying on variable names, to the interprocedural detection algorithm. Other, non-obvious, issues become relevant. For example, it is important to know which method can be invoked as a result of a given call; this implies that accurate pointer analysis, which reduces the potential candidates, can be an important tool for clone detection. Increasing use of semantic information at the expense of syntax also creates artifacts such as that of Figure 7, in which an interprocedural clone exists at the implementation level but is not apparent as such in the text.

Cider is a step in the direction of clone detection in the presence of ubiquitous refactoring tools. We believe that the plan representation, which abstracts away many syntactic details and is canonical to a large extent, is useful as the basis of such clone detectors. Some features of the plan representation eliminate the effect of certain refactorings; as mentioned in Section 3.1, the plan does not change when variables are renamed, or when Extract Local Variable (or its inverse, Inline Variable) is performed, since data flow is expressed by data edges rather than by name. Changing the syntactic form in which control flow is expressed, such as replacing a a `for` loop by a `while` form, likewise preserves the plan form.

Other, more global, changes do affect the plan representation, and the clone-detection algorithm needs to cater to them explicitly. The Cider algorithm shown in Section 3 finds interprocedural clones, by following data and control paths across calls and returns. The fact that data flow is represented by edges considerably simplifies the algorithm, which can ignore the difference between data flow through local variables or through method parameters and return values. As mentioned above, information about refactorings that are likely to have been performed can help Cider identify clones more precisely; for example, RefactoringCrawler [21] can suggest methods that have been renamed, providing seeds for the Cider algorithm to search from. It is interesting to note that Cider can also be used to enhance RefactoringCrawler by allowing it to discover instances of Extract Method and Inline Method.

The use of the plan representation also makes Cider easy to apply to other programming languages. Cider is built on top of IBM Research's implementation of the plan representation, and can therefore be applied to any language for which a path to the plan exists.

The Cider algorithm can be applied to other representations, but may require extending them in various ways. The interprocedural extension of the PDG is the System Dependence Graph (SDG) [24]. As we noted earlier, the fine-grained representation of operations in the plan causes some refactorings (such as Extract Local Variable) to have no effect on the plan representation. Recognizing such refactorings with the SDG will require modifying it appropriately; for example, Krinke [11] used a "fine-grained PDG" to find intraprocedural clones. The SDG does not normally contain information about ordering between statements in the same control block. This makes it difficult to recognize those clones that contain consecutive but unrelated statements. Whether or not this is desirable

depends on the specific application; in any case, using the plan representation it is possible to ignore this information or use it to filter possible matches. Both Krinke [11] and Higo and Kusumoto [25] extended the PDG with dependence edges that correspond to the plan's control-flow edges. We also believe that the explicit interprocedural data- and control-flow edges in the plan will make it easier to implement various kinds of path-matching heuristics, as we discuss next.

Cider can be extended in various ways in order to discover clones that have been even further modified. Because the Cider algorithm does not require a simultaneous match of both data and control, it is able to follow data edges in spite of other code that has been inserted or removed in the control path between the source and target of the edge. Similarly, it can follow control edges even when data sources have changed. As discussed in Section 3.2, Cider can find clones even when parts are embedded inside conditionals in one case but not the other. The use of an expression in the conditional can confuse the current implementation when other clues (such as data flow) are not available. However, the algorithm can be modified to bypass the condition in its search for matching data or control paths. More generally, it can also be extended to match paths even when interior nodes do not match. As mentioned in Section 3.2, Cider matches *flows*, defined to be paths that start and end in computational nodes and only go through intermediary nodes. It is possible to allow computational nodes inside flows as well, thus bypassing some computations in the match. For example, the addition of a negation operator into part of a boolean expression will result in an additional computational node for that operation in the plan. By matching the data path that includes that new node to the original edge, the algorithm can discover the similarity in spite of the change. Of course, this can lead to combinatorial explosion if not done carefully; we are now investigating how to use this technique effectively to find more clones.

As mentioned above, there are various parameters of the algorithm we need to investigate. The size thresholds on identified clones, which determine which fragments will be classified as clones, have an obvious affect on both recall and precision. The similarity criteria that determine which nodes can be matched also have a significant effect on the clones that can be detected and the refactorings that can be overcome in the process. Finally, better seeds will obviously result in better clones, and the generation of good seeds is an important direction for future research.

Syntax-based clone detectors, especially those based on k-gram matching [5], can scan large code bases efficiently in order to find pairs of matches across the whole corpus. Semantic detectors like Cider can never hope to match this efficiency. As a result, Cider needs to use other, faster, methods to discover seeds. The method used to find the initial seeds affects the operation of Cider in two ways: more seeds can improve recall, while fewer seeds improve efficiency. Good heuristics for finding seeds can improve both aspects, and merit further research.

As mentioned in the introduction to this paper, we have previously developed the Extract Computation refactoring [4], which is a significant generalization of

Extract Method that is able to extract non-contiguous pieces of code, and split loops, including the generation of data structures to pass information from one incarnation of the loop to the next. The code to be extracted can be specified manually, but it can also come from some other tool. Specifically, we are interested in clone remediation by abstracting common parts of two or more clones. By comparing the semantic structure of clones found, we hope to be able to automatically identify those common parts, and then use Extract Computation to perform the remediation.

Acknowledgments. We are grateful to Amiram Yehudai for helpful discussions and encouragement.

References

1. Balint, M., Marinescu, R., Girba, T.: How developers copy. In: Proc. 14th IEEE Int'l Conf. Program Comprehension (ICPC 2006), pp. 56–68 (2006)
2. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2000)
4. Abadi, A., Ettinger, R., Feldman, Y.A.: Fine slicing: Theory and applications for computation extraction. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 471–485. Springer, Heidelberg (2012)
5. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: Proc. 2003 ACM SIGMOD Int'l Conf. Management of Data (SIGMOD), pp. 76–85 (2003)
6. Jia, Y., Binkley, D., Harman, M., Krinke, J., Matsushita, M.: KClone: A proposed approach to fast precise code clone detection. In: Proc. Third Int'l Workshop on Software Clones, IWSC (2009)
7. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: Scalable and accurate tree-based detection of code clones. In: Proc. 29th Int'l Conf. Software Engineering (ICSE), pp. 96–105 (2007)
8. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Engineering 28(7), 654–670 (2002)
9. Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance, pp. 368–377 (November 1998)
10. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001)
11. Krinke, J.: Identifying similar code with program dependence graphs. In: Proc. Eighth Working Conference on Reverse Engineering (WCRE 2001), pp. 301–309 (2001)
12. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: Proc. 30th Int'l Conf. Software Engineering (ICSE), pp. 321–330 (2008)
13. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. of Comp. Prog. 74(7), 470–495 (2009)

14. Soares, G., Catao, B., Varjao, C., Aguiar, S., Gheyi, R., Massoni, T.: Analyzing refactorings on software repositories. In: Proc. 25th Brazilian Symp. Software Engineering (SBES), pp. 164–173 (2011)
15. Liu, C., Chen, C., Han, J., Yu, P.S.: GPLAG: Detection of software plagiarism by program dependence graph analysis. In: Proc. 12th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD), pp. 872–881 (2006)
16. Rich, C., Waters, R.C.: The Programmer's Apprentice. ACM Press and Addison Wesley (1990)
17. Rich, C.: A formal representation for plans in the Programmer's Apprentice. In: Proc. 7th Int. Joint Conf. Artificial Intelligence, Vancouver, British Columbia, Canada, pp. 1044–1052 (August 1981)
18. Feldman, Y.A., Friedman, D.A.: Portability by automatic translation: A large-scale case study. Artificial Intelligence 107(1), 1–28 (1999)
19. Cohen, Y., Feldman, Y.A.: Automatic high-quality reengineering of database programs by abstraction, transformation, and reimplementation. ACM Trans. Software Engineering and Methodology 12(3), 285–316 (2003)
20. 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)
21. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 404–428. Springer, Heidelberg (2006)
22. Weiser, M.: Program slicing. IEEE Trans. Software Engineering 10(4) (July 1984)
23. Godfrey, M.W., Zou, L.: Using origin analysis to detect merging and splitting of source code entities. IEEE Trans. Software Engineering 31, 166–181 (2005)
24. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12(1), 26–60 (1990)
25. Higo, Y., Kusumoto, S.: Code clone detection on specialized PDGs with heuristics. In: Proc. 15th European Conf. Soft. Maintenance and Reengineering, CSMR (2011)

A Compositional Paradigm of Automating Refactorings

Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson

University of Illinois at Urbana-Champaign
{mvakili2,nchen,rzilouc2,snegara2,rjohnson}@illinois.edu

Abstract. Recent studies suggest that programmers greatly underuse refactoring tools, especially for *complex* refactorings. Complex refactorings tend to be *tedious* and *error-prone* to perform by hand. To promote the use of refactoring tools for complex changes, we propose a new paradigm for automating refactorings called *compositional refactoring*. The key idea is to perform small, *predictable* changes using a tool and *manually* compose them into complex changes. This paradigm trades off some level of automation by higher *predictability* and *control*. We show that this paradigm is *natural*, because our analysis of programmers' use of the Eclipse refactoring tool in the wild shows that they frequently batch and compose automated refactorings. We then show that programmers are *receptive* to this new paradigm through a survey of 100 respondents. Finally, we show that the compositional paradigm is *effective* through a controlled study of 13 professional programmers, comparing this paradigm to the existing wizard-based one.

1 Introduction

Refactoring is defined as changing code without affecting the observable behavior of the program [6, 9, 21]. Refactoring is not only recommended by expert practitioners [6, 11], but also commonly practiced by programmers [17, 19, 27, 34]. The first refactoring tool was invented more than a decade ago to make the refactoring process faster and more reliable [23]. Today, modern Integrated Development Environments (IDEs), such as Eclipse, IntelliJ IDEA, NetBeans, and Xcode, support many refactorings that rename, move, split, or join various program elements including methods, classes, and packages. In addition, researchers continue to propose new tools for automating complex changes [4, 13, 25, 29, 33].

Despite the expected benefits of automated refactorings, studies have shown that programmers greatly underuse these tools, especially for complex changes. Although complex refactorings are more *tedious* and *error-prone* than simple ones to perform manually, programmers use the refactoring tools mostly for performing simple refactorings such as Rename, Extract Local Variable, and Extract Method [19, 20, 27].

The mainstream refactoring tools follow a wizard-based paradigm. Typically, a programmer selects a piece of code in the editor and invokes an automated refac-

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 527–551, 2013.

toring from a menu. The programmer may then change some of the configuration options on the wizard. These options control the outcome of the refactoring by specifying the entities that should be created, copied, or moved. The tool may also preview the change, e.g., by showing snapshots of the affected files before and after the refactoring side-by-side.

Prior studies have identified several problems with the wizard-based paradigm of refactoring [19,27]. For instance, the long list of automated refactorings in the menu leads to higher *learning* and *invocation* costs. The context-switch from a code editor to a wizard *disrupts* the programming workflow. The wizard imposes an upfront *configuration* cost, making it difficult to *control* the outcome of the tool. The preview page of the wizard is often too cluttered, which makes the refactoring tool less *predictable*. That is, the programmer cannot easily predict how the tool is going to affect her code. Even if a programmer makes her way through all the steps of invocation, configuration, and preview, the wizard may still notify her at the end that the refactoring is impossible or unsafe to perform. These problems call for rethinking the design of refactoring tools.

The main contribution of this paper is a new paradigm, called *compositional refactoring*, for automating complex refactorings. The key idea is to have the tool automate small, predictable changes and let the programmer manually compose these changes into complex ones. For instance, rather than performing a large refactoring such as Extract Superclass in a single step, the compositional paradigm automates small steps, e.g., Create New Superclass and Move Member to Immediate Superclass, leaving it to the programmer to compose these steps.

The compositional paradigm offers a lower level of automation than the wizard-based one by automating small changes. It puts the programmer in control by letting her compose the small changes. Although it may seem counterintuitive that reducing the level of automation improves an automated tool, this phenomenon is not new. Other fields such as aviation, health-care, and manufacturing have gone through a similar process. Motivated by the perceived benefits of automation, highly automated systems were invented, often neglecting the role of the human operators. Further studies showed that often a less automated system with a human-centered design performs better, concluding that *less is (sometimes) more*, when it comes to automation [2,12,31].

The idea of compositional refactoring is inspired by our studies of the refactoring practices in the wild. Even though expert practitioners recommend that programmers perform refactorings as a composition of smaller ones [6,11], little is known about how programmers compose refactorings in practice. Therefore, we mined two refactoring data sets: the *Eclipse foundation* and *Illinois* data sets. The Eclipse foundation has collected data from hundreds of thousands of programmers over the years. Our data mining of this large corpus of data shows that programmers *frequently* invoke a *variety* of multiple automated refactorings within a short period of time. Nonetheless, refactorings invoked in close time proximity may be semantically unrelated. Therefore, we consulted the Illinois data set, which we collected during a prior field study from 30 programmers over eight months. The Illinois data set is smaller but contains more contextual

information about refactoring invocations. We manually inspected a sample of its refactorings that were invoked in a short time span. This analysis reveals some of the *rationales* for *systematically* composing refactorings, providing further evidence for the *naturalness* of the compositional paradigm to programmers.

We evaluated the idea of compositional refactoring in two ways. We first distributed an online survey to get early feedback from hundreds of programmers on our design (Section 5). The survey presented mockup screenshots of compositional and wizard-based refactorings and asked the participants to compare the two paradigms. The survey results showed that programmers are *receptive* to the idea of compositional paradigm and provided *improvement suggestions*. This positive response motivated us to implement the compositional paradigm.

We enhanced the design based on the feedback we received from the survey participants. Then, we implemented it as an Eclipse plug-in. Finally, we conducted a lab study with 13 professional programmers at a large software company (Section 6). We instructed the participants to perform a refactoring on an open-source project using the compositional and wizard-based refactoring tools in a random order. Like the survey participants, the majority (nine) of the lab study participants were more satisfied with the compositional paradigm than the wizard-based one. In addition, the participants were more likely to finish the task *correctly* and significantly *faster* in the compositional paradigm.

Overall, the participants of the survey and lab studies appreciated the compositional paradigm because of its perceived higher level of control, easier method of invocation, and interactivity. In addition, they suggested features like abstract views and multi-selections. These results suggest that compositional refactoring is a promising paradigm for assisting programmers in performing complex refactorings. Our work contributes to the refactoring practice in several ways:

1. We provide empirical evidence for the prevalence and nature of automated refactorings that are invoked in close time proximity (Section 2).
2. We discuss some of the rationales for composing automated refactorings based on our manual inspection of the Illinois data set (Section 3).
3. We propose a new paradigm for automating complex refactorings, namely compositional refactoring (Section 4).
4. We provide an implementation of compositional refactoring as an Eclipse plug-in (Section 6.1).
5. We show the effectiveness of compositional refactoring through a survey (Section 5) and a lab study (Section 6).
6. We draw implications from our analyses of refactoring usage data, survey study, and lab study for designing future tools that better support complex refactoring.

Our tool and study artifacts are available at `http://codingspectator.cs.illinois.edu/compositional-refactoring`. These artifacts have been successfully evaluated by the ECOOP artifact evaluation committee and found to meet expectations.

2 Frequent Refactoring Sets

In this section, we answer the following research questions:

- How *frequently* do multiple kinds of refactorings occur in a short time span?
- How *diverse* are the refactorings frequently invoked in a short time span?

Answers to these questions provide a bird's-eye view of the phenomenon of invoking several automated refactorings in a short time span.

2.1 Eclipse Foundation Data Set

Usage Data Collector (UDC) is a pre-installed plug-in in Eclipse, which records uses of Eclipse commands, views, and perspectives. UDC generates a fresh identifier for the user and persists it in the home folder of the user. For each event or action performed by the user, UDC captures the timestamp, the event identifier, the user identifier, and the bundle that generated the event. If a user agrees, UDC regularly sends the user's data to the Eclipse foundation's servers. We analyzed a subset of the UDC data that contained information about the invocations of the Eclipse refactoring tool for Java. The Eclipse foundation has released the data from a total of 195,105 programmers who used the Eclipse refactoring tool for Java during 20 months from January 2009 until August 2010.

2.2 Data Analysis

We used the large data set of Eclipse foundation to infer the *frequent refactoring sets*, the sets of automated refactorings that are frequently invoked in a short time span. Since the refactorings invoked in temporal proximity may not be semantically related, this analysis only provides a bird's-eye view of the frequency and variety of compositions of automated refactorings in the wild.

Refactoring Batches. Intuitively, a *refactoring batch* is a maximal set of automated refactorings, such that the consecutive refactorings are invoked within a close time proximity. A refactoring batch is a nonempty set of refactoring kinds. For instance, the refactoring batch {Move, Rename} may stand for one or more invocations of Move and Rename in any order within a close time proximity.

We partitioned the refactoring events into refactoring batches using a heuristic. The heuristic uses the large gaps between the invocation times of consecutive refactorings as the partition boundaries. This heuristic is based on the assumption that refactorings invoked far apart in time are less likely to be semantically related. First, the partitioning algorithm sorts the refactoring events of every UDC user by invocation time. Next, the algorithm creates a new batch for each user and adds the kind of the first refactoring event of the user to the batch. If a refactoring event is invoked by the same user within δ minutes of the preceding event, the algorithm will add the kind (Rename, Move, etc.) to the batch of the preceding event. Otherwise, the algorithm will add the kind to a new batch. When the batch of every refactoring event is determined, the algorithm terminates and returns the set of refactoring batches.

Mining Frequent Refactoring Sets. A *refactoring set* is a nonempty subset of a refactoring batch. The *support* of a refactoring set R is the fraction of refactoring batches that are supersets of R. We applied a frequent itemset mining algorithm [10, pp. 246–50] on the set of refactoring batches to infer the *frequent refactoring sets*—the refactoring sets with the highest supports.

We used an implementation of the frequent itemset mining algorithm in the statistical computing software R [1]. We provided the algorithm with refactoring batches ($\delta = 10$) and set the parameter `minsup` to 0.001. The output of the algorithm is a list of refactoring sets with a support of at least `minsup`.

We repeated the analysis for each $\delta \in \{5, 10, 20, 40\}$, and compared the resulting frequent refactoring sets. Due to the negligible effect of such changes of δ on the most frequent refactoring sets, we present the results for only $\delta = 10$.

Table 1. The 20 most frequent refactoring sets of UDC active users. A *refactoring batch* is the kinds of a set of automated refactorings such that the consecutive refactorings are invoked within 10 minutes. A *refactoring set* is a subset of a refactoring batch. For instance, the refactoring set {Pull Up} stands for one or more invocations of Pull Up in a refactoring batch. The support of a refactoring set R is the fraction of batches that are supersets of the R.

refactoring set	support
{Rename}	0.591
{Extract Local Variable}	0.270
{Extract Method}	0.154
{Inline}	0.090
{Extract Local Variable, Rename}	0.076
{Move}	0.058
{Extract Method, Rename}	0.057
{Change Method Signature}	0.055
{Extract Constant}	0.043
{Extract Local Variable, Extract Method}	0.042
{Inline, Rename}	0.033
{Extract Local Variable, Inline}	0.031
{Extract Method, Inline}	0.027
{Convert Local Variable to Field}	0.025
{Move, Rename}	0.024
{Change Method Signature, Rename}	0.022
{Extract Local Variable, Extract Method, Rename}	0.020
{Pull Up}	0.016
{Extract Local Variable, Inline, Rename}	0.015
{Extract Constant, Rename}	0.015

2.3 Results

The data mining algorithm inferred 47 frequent refactoring sets for all UDC users. However, the vast majority of UDC users use automated refactorings rarely. Most (98.6%) users invoked the refactoring tool at most 50 times, and 98.9% invoked at most five kinds of automated refactorings. We consider users who invoked at least five kinds of automated refactorings for a total of at least 50 times *active* and the rest *inactive*. This leads to 1,188 active users with a total of 112,885 refactoring events.

We hypothesized that the data of inactive users conceals some of the frequent refactoring sets of the active ones. Thus, we repeated the data mining algorithm on the active users alone. This resulted in about three times more frequent refactoring sets ($N = 150$), 44 of which were also inferred for all users. This indicates that limiting the data set to active users uncovers more frequent refactoring sets. Table 1 lists the most frequent refactoring sets of active UDC users. This table shows the following:

1. Programmers invoke a variety of automated refactorings in a short time span.
2. Some refactoring sets with multiple refactoring kinds are more frequent than those with a single kind. For instance, {Extract Local Variable, Extract Method} is about 2.5 times more frequent than {Pull Up}. In other words, a refactoring batch is more likely to contain Extract Local Variable and Extract Method than at least one Pull Up. This result reveals a limitation of prior studies [17, 19, 27], which focused only on individual refactorings.

3 Refactoring Composition Patterns

The frequency and variety of refactoring sets (Section 2.3) led us to the hypothesis that programmers *systematically compose* certain kinds of automated refactorings to apply larger changes. This section presents answers to the following reseach questions:

- Do programmers compose automated refactorings?
- What are some of the rationales for composing automated refactorings?

The analysis of the Eclipse foundation data set showed that certain kinds of automated refactorings (e.g., {Extract Local Variable, Extract Method}) are frequently invoked in a short time span. Although this data set is huge, it does not capture enough context about each event to infer the rationales for invoking several automated refactorings in a short time span. Therefore, we analyzed the smaller but more detailed Illinois data set.

3.1 Illinois Data Set

The Illinois data set comes from two of our Eclipse-based data collectors, namely *CodingSpectator* and *CodingTracker* [27, 28]. CodingTracker records applications of all 33 automated refactorings of Eclipse, and CodingSpectator records more detailed data (e.g., the piece of code surrounding the refactored program element and error messages reported by the refactoring tool) for 23 automated refactorings.

The Illinois data set contains data from 30 programmers consisting of a total of 2,296 programming hours over eight months. Fourteen of our participants were external developers who we recruited by sending invitation messages to individual programmers, mailing lists, and IRC channels of open-source projects. We also recruited twelve graduate students and four interns from six research

labs at the computer science department of the University of Illinois at Urbana-Champaign. Based on the results of our demographic survey that 28 participants took, 1, 5, 15, and 7 participants had 1–2, 2–5, 5–10, and more than 10 years of programming experience, respectively.

3.2 Data Analysis

The partitioning algorithm (Section 2.2) found 1,633 refactoring batches of 244 kinds in the Illinois data set. We selected 32 kinds of batches, which were frequent in the Illinois data set or contained the frequent refactoring sets of the Eclipse foundation data set. Then, we manually analyzed a random sample of at most ten batches of each kind, leading to a total of 139 batches.

We examined the information captured for each refactoring event in a batch, e.g., the kind, invocation time, error messages, and the piece of code surrounding the selection. Based on these data, we decided if the refactorings in the batch were semantically related, and inferred a rationale for the batch. Next, for each batch kind, we collected the rationales of the batches of that kind. Finally, we collected five *refactoring composition patterns*. A refactoring composition pattern is a recurring set of automated refactorings that programmers compose for similar rationales.

3.3 Results

We found that the majority (81%, i.e., 112 of 139) of the analyzed batches contained related refactorings. The following presents the refactoring composition patterns that we observed in our sample of refactoring batches. Each pattern reveals some of the *rationales* for composing refactorings, providing evidence that programmers *systematically* compose automated refactorings. The value of n in parentheses shows the number of refactoring batches with a particular property.

Refactoring Closely Related Entities ($n = 47$). We found that programmers frequently compose refactorings to refactor closely related entities that are not related by name binding. For instance, the participants composed several Rename refactorings on program entities with similar names ($n = 8$) or a method and the variable that gets the return value of the method ($n = 2$). As another example, our participants performed the Rename refactoring to rename a field and the constructor parameter that initialized the field ($n = 2$).

Refactoring tools only update the entities that are syntactically related. For instance, the Rename refactoring updates the declaration and all references of a name. One recommendation for future tools is to support this refactoring composition pattern by reliably detecting the names that are likely to co-evolve.

Adapting Extract Method ($n = 34$). We found that programmers compose three kinds of refactorings, Extract Local Variable, Extract Method, and Inline Local Variable, to adapt the outcome of Extract Method (Figure 1). This refactoring composition pattern consists of three steps: *preparation*, *method extraction*, and *simplification*. First, the programmer performs Extract Local Variable

```
public static void main(String[] args) {
  int factorial = 1;
  for (int i = 1; i <= 10 ; ++i) {
    factorial *= i;
  }
  System.out.println(factorial);
}
```

(a) The initial code. The programmer extracts `10` into a new local variable `n`.

```
public static void main(String[] args) {
  int factorial = 1;
  int n = 10;
  for (int i = 1; i <= n; ++i) {
    factorial *= i;
  }
  System.out.println(factorial);
}
```

(b) The programmer moves the declaration of `n` to exclude it from her future selection.

```
public static void main(String[] args) {
  int n = 10;
  int factorial = 1;
  for (int i = 1; i <= n; ++i) {
    factorial *= i;
  }
  System.out.println(factorial);
}
```

(c) The programmer extracts the computation of factorial of `n` into a new method.

```
public static void main(String[] args) {
  int n = 10;
  int factorial = getFactorial(n);
  System.out.println(factorial);
}

private static int getFactorial(int n) {
  int factorial = 1;
  for (int i = 1; i <= n; ++i) {
    factorial *= i;
  }
  return factorial;
}
```

(d) The programmer inlines local variable `n`, which is now used just once.

Fig. 1. Participants composed Extract Local Variable, Extract Method, and Inline Local Variable to extract methods with their desired signatures

so that the upcoming Extract Method refactoring infers a method parameter corresponding to the extracted local variable. Second, she invokes Extract Method on a piece of code excluding the declarations of any variables added during the preparation step. Finally, the programmer invokes Inline Local Variable to simplify the code. A refactoring batch with method extraction may contain only the preparation step ($n = 11$), only the simplification step ($n = 19$), or both ($n = 4$). It is impossible to configure the refactoring wizard of Extract Method to extract the same method in one step.

Instead of composing three refactorings to include certain parameters in the signature of the extracted method, the programmer could compose just two refactorings, namely, Extract Method and Introduce Parameter. However, there were no instances of the latter in the Illinois data set. In general, the automated Introduce Parameter refactoring is used infrequently and fewer programmers know about it [19, 27]. Nonetheless, a programmer can adapt Extract Method without the need to learn the Introduce Parameter refactoring.

The following are some of the rationales of this composition pattern:

- configuring a refactoring in ways not supported by a wizard
- avoiding the need to learn additional kinds of automated refactorings

If a refactoring tool is aware of the composition patterns for adapting a refactoring, the tool could offer to perform the simplifications in one step.

Backtracking Refactorings ($n = 12$). Not all refactorings in a batch contribute to the overall effect of the batch. For example, a pair of Extract Local Variable and Inline Local Variable refactorings may cancel the effects of each other ($n = 6$). A refactoring batch may also contain a refactoring that is later undone ($n = 6$). For example, we found that the participants extracted a piece of code into a local variable, undid the refactoring, and finally extracted the same piece of code into a constant ($n = 2$). This indicates that programmers experiment with refactorings or accidentally invoke the wrong refactoring.

Composition-over-configuration ($n = 8$). *Composition-over-configuration* is a composition pattern that we found programmers employ to avoid the upfront configuration cost of refactoring wizards. With this pattern, the programmer composes multiple automated refactorings to perform a refactoring that could have been done by configuring a refactoring wizard.

For example, it is possible to configure the Pull Up refactoring wizard to move one or more members (fields or methods) of a class to a superclass in one step. However, the participants sometimes composed two Pull Up refactorings to pull up two members of a class one at a time ($n = 2$).

As another example, the Extract Local Variable refactoring wizard allows the programmer to set the name of the new variable. Nevertheless, a participant composed Extract Local Variable by a Rename ($n = 1$).

Observing the composition-over-configuration pattern, we propose the compositional paradigm of automating refactorings (Section 4). We implemented the compositional paradigm using a feature of Eclipse called *Quick Assist* (Figure 2). Quick Assist is a popular method of invoking refactorings that supports composition-over-configuration [27]. For example, if a programmer invokes Extract Method through Quick Assist, it would apply Extract Method with a default name and then initiate a composition with Rename on the method name.

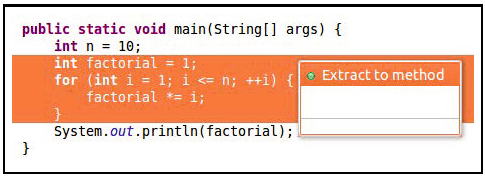


Fig. 2. A screenshot of Eclipse Quick Assist (`CTRL+1`). In this case, Quick Assist suggests Extract Method as a refactoring applicable to the selected piece of code.

Multiple Refactorings on an Entity ($n = 6$). A program entity may undergo multiple refactorings. For instance, the participants composed Extract Method with Pull Up on the same method to refactor to the Template Method design pattern ($n = 2$) [7, p. 325]. As another example, a participant composed Push Down with Encapsulate Field on the same field ($n = 1$).

4 Design of a Compositional Refactoring Tool

Based on the lessons learned from our data analysis, literature review, and our prior research studies, we compiled a list of design goals for a new refactoring tool. These goals inspired the design and implementation of a tool for compositional refactoring. The tool currently supports several refactorings such as Extract Superclass, Extract Interface, Pull Up, and Push Down, and can be extended to support other refactorings. In this section, we first discuss our design goals and then explain the steps involved in performing the Extract Superclass refactoring using our tool.

4.1 Design Goals

Predictability. Our prior study showed that programmers rarely use the automated refactorings whose outcomes are not easily predictable [27], e.g., the refactorings that affect several files. Our goal is to make such refactorings more predictable. One strategy to achieve predictability, employed by the wizard-based refactorings, is to assist the programmer in reviewing the changes. Another strategy, which we have explored in our compositional paradigm, is to divide a large refactoring into smaller, predictable refactorings.

Control. Programmers prefer to maintain control over the evolution of their code during a refactoring [27]. One way to control a refactoring is to allow the programmer to configure it upfront. However, configuration dialogs increase the cost of using the tool [19, 27], and programmers rarely configure the refactoring tool [19]. An alternative paradigm is to put the programmer in control by assisting her in performing the refactoring in smaller steps. In this paradigm, she can evaluate the refactoring at each step and intersperse it with manual edits.

Discoverability. Researchers and tool vendors continue to automate more recurring code transformations [4, 13, 25, 29, 33]. However, programmers discover only a subset of the automated refactorings [27]. Quick Assist makes the refactorings more discoverable by proposing them based on the current context. Programmers frequently use Quick Assist to discover and invoke refactorings [27]. Thus, our tool relies on Quick Assist as its method of invocation.

Learnability. The list of automated refactorings in modern IDEs is long. The cost of learning so many tools is a barrier to their adoption. Our goal is to solve this problem by allowing programmers learn a small number of *reusable* refactorings and compose them to perform many kinds of larger refactorings.

Low Disruptiveness. Although configuration dialogs make the refactoring tool more powerful and customizable, they distract the programmer from the code and disrupt her flow of programming [19, 27]. We aim to design refactoring tools that are highly interactive and allow the programmer to focus on the code.

Correctness. Wizard-based refactorings guarantee correctness by checking a set of *preconditions*. Similarly, compositional refactorings check the preconditions of the individual steps. Because the steps are small, we expect that programmers can verify them more easily. Moreover, the programmer can run tests after each step. This allows the programmer to identify the exact step that led to a problem.

4.2 Compositional Extract Superclass Refactoring

Our goals informed the design of a compositional refactoring tool. We use the Extract Superclass refactoring as an example to demonstrate the compositional paradigm. This refactoring lets the programmer create a superclass for one or more classes and move some of the members of the subclasses to the superclass. We chose Extract Superclass because it is one of the more complex and less frequently used automated refactorings of Eclipse [19, 27]. Figure 3 shows our mockup of compositional Extract Superclass. We later improved the mockup and implemented it as an Eclipse plug-in (Section 6.1). In the following, we briefly describe how the tool works.

1. The programmer selects the class (`Daisy`) to extract a superclass from.
2. She selects "Create new superclass in file" from the Quick Assist menu.
3. This creates a new empty superclass and prompts for a new name (`Flower`).
4. The programmer invokes "Move to immediate superclass" on method `water`.
5. This moves method `water` from `Daisy` to `Flower`.
6. The programmer invokes "Move type to new file" on class `Flower`.
7. This moves class `Flower` to a new file and completes the refactoring.

The outcome of each of the above steps is immediately visible to the programmer in the code editor. At each step, the Quick Assist menu suggests a set of actions that are applicable to the selected element. The steps are *independent* of each other. That is, Quick Assist suggests the steps regardless of what step was previously performed. The programmer does not have to perform every step using our Quick Assist actions. Rather, she can perform some steps manually.

Fig. 3. Mockup screenshots of compositional Extract Superclass. See Section 4.2 for a description of each screenshot. The survey used similar screenshots (Section 5). We later implemented the mockup as an actual tool (Section 6.1).

5 Survey Study

We distributed a survey to assess our design goals and compare our compositional prototype of the Extract Superclass refactoring (Figure 3) with the existing wizard-based user interface of this refactoring in Eclipse. The goal of the survey study is to answer the following research questions:

- How do programmers compare the compositional and wizard-based paradigms?
- Are programmers likely to adopt the compositional paradigm?
- What are some opportunities for improving the compositional paradigm?

Answering these questions shows how *receptive* programmers are to the new compositional paradigm.

5.1 Method

We recruited 100 programmers by announcing the survey[1] on `reddit.com`[2], `twitter.com`, and mailing lists of open source projects. The survey was estimated to take 20 minutes, and started with questions about the experience of the respondent with programming, IDEs, and refactoring. Then, it asked about the programmer's strategy in performing the Extract Superclass refactoring. Finally, the survey presented screenshots of the two user interfaces of the Extract Superclass refactoring, and asked the respondent to evaluate and compare them.

5.2 Thematic Coding

We employed *thematic coding* [32], a systematic qualitative method, to analyze the responses to open-ended questions. The coding was *inductive* (data-driven) as opposed to *deductive* (theory-driven). We extracted the opinions and ideas associated with each segment of the comments. Through an iterative process, we defined, merged, and split the themes to better identify the central ideas. The goal of such a coding is to identify the major ideas not to count the frequencies of keywords. This coding allowed us to reliably decide if two participants provided equivalent responses. For each statement that we report, we include the number of participants that agree with it as an indication of its overall support.

5.3 Participants

The participants were familiar enough with modern programming environments to evaluate the compositional paradigm. The majority (91%) of the survey respondents had more than five years of programming experience. Of all the participants, 76% considered themselves to be experts in at least one programming language (on a five-point Likert scale ranging from "Unfamiliar" to "Expert"), and 99% rated themselves as either very familiar with or expert at one or more languages. The respondents indicated that they were familiar with Eclipse (81%), Visual Studio (58%), NetBeans (39%), IntelliJ (36%), and Xcode (28%).

5.4 Results

The survey presented screenshots of the steps to perform the Extract Superclass refactoring using both the compositional and wizard-based paradigm on the same page. The survey asked the participant how often she would use each interface on a five-point Likert scale ranging from "Never" to "Nearly every time". The majority (66%) of respondents said that if both compositional and wizard-based paradigms are available, they would use the compositional paradigm at least as frequently as the wizard-based one. More interestingly, those who did not use an existing tool for Extract Superclass or used the tool some of the time were

[1] `https://illinois.edu/fb/sec/8454746`
[2] `http://www.reddit.com/r/programming`

more likely to prefer the compositional paradigm (Table 2). This shows that the compositional paradigm is a promising technique for increasing the utilization of automated refactorings.

Table 2. Joint distribution of respondents' frequency of using the Extract Superclass refactoring wizard and their preferred paradigm (compositional vs. wizard-based). Since three respondents did not indicate their frequency of using the wizard, the last row is slightly different from the sum of the other rows.

	Prefers composition	**Has no preference**	**Prefers wizard**
Does not use the wizard	21%	4%	6%
Sometimes uses the wizard	16%	5%	9%
Uses the wizard	15%	3%	20%
All respondents	52%	14%	34%

Finally, the survey asked the participants to compare and evaluate the paradigms. We applied a qualitative analysis method (Section 5.2) on the comments provided by 50 participants. The following discusses the result of this analysis, which reveals the *strengths* and *weaknesses* of each paradigm and highlights opportunities for *improvement*.

Control. Three survey respondents indicated that they would prefer the compositional paradigm, because it provides more control over the evolution of code. For instance, P_5 wrote:

> I think the second [compositional] UI [...] gives me the idea of having the control over what's happening, and how further can I go with it.

This result is consistent with the findings of a prior study, which showed that programmers prefer to maintain control over their code and use automated refactorings whose outcomes are predictable [27].

Invocation Method. Two respondents reported that it was difficult to invoke the wizard-based refactorings from the menu mostly because the menu was too cluttered. Five said that they would prefer keyboard shortcuts. For instance, P_{10} suggested the following as a way to improve the wizard-based interface:

> Make refactoring initiated by keyboard short-cut and not buried so deeply in a menu.

However, as one respondent said, keyboard shortcuts are hard to remember. Quick Assist is a middle-ground, because it is keyboard navigable and proposes only the refactorings that are applicable to the current context. Two participants said that Quick Assist was an easier way of invoking refactorings.

Incrementality and Testability. Six respondents mentioned that they do not want refactoring tools with modal dialogs. Five said that modal dialogs are distracting, and three said that the dialogs are too complex. On the contrary, five people indicated that the compositional paradigm is more interactive and two indicated that it allows running tests after each step. For instance, P_7 said:

> The second [compositional] one provides a more stepwise view, giving me more intermediate feedback, as well as an ability to run my tests at each step. This goes a long way to making sure the refactoring is the right decision.

Nonetheless, one respondent said that the compositional paradigm had too many steps, and two preferred to perform the refactoring in a single step. For example, P_8 said:

> I don't write Hello World examples. I need control over what gets moved up and what not, what's made abstract and so forth. I want to do this in one pass, not six [four].

We made compositional paradigm incremental to achieve high control and testability. We decided that compositional refactoring fits programmers' workflow, because it mimics manual refactorings and programmers already compose refactorings (Section 3).

Abstract View. Nine participants were concerned that the compositional paradigm may not be suitable for large refactorings. For example, P_2 said:

> [I'm] Not sure I'd want to use that [wizard-based] UI for any refactoring work. However, [the wizard-based UI is] probably better for very large refactoring tasks than the second [compositioanal] UI—but if you're doing that, you're doing too much in one go.

Four respondents said that a high-level view of the code would be useful for performing large refactorings. For instance, P_3 said:

> [...] However, specifying the methods to be moved one by one rather than selecting them from a list might cause methods that should be extracted into a superclass to be missed. In some sense, it is mostly about whether an abstract view of the methods is preferable to a code level view when choosing whether to extract them. Sometimes I find myself leaving the extract superclass dialog to figure out what a method actually does and whether it should be extracted.

The wizard-based paradigm lets the programmer operate at the level of classes and methods, but, makes it difficult to switch between the code and its higher-level view. On the other hand, the compositional paradigm that we demonstrated on the survey was tightly coupled with the code, which makes it easy to intersperse low-level code changes with refactorings. To offer the benefits of both, the tool could make the switch between the two views seamless, e.g., by making the refactorings available both in the code editor and graphical views.

Multi-selections. Five respondents preferred to be able to select multiple program entities and manipulate them at the same time. For example, P_4 said:

> Usually, I'm extracting a common superclass to remove duplication from more than one similar class, so I'd need to be able to select multiple classes.

Extending the compositional paradigm to support an abstract view will make it possible to select multiple program elements from the abstract view in one step.

Coding Conventions. The mockups of the compositional paradigm showed how to first create an empty superclass in the same file and move it to a new file later (Figure 3). Although one could move the superclass to a new file right after creating the superclass, nine preferred the tool to adhere to the coding conventions strictly and never introduce two classes in the same file.

6 Lab Study

The survey study showed the overall preference of programmers towards compositional refactoring based on participants' evaluations of the mockup screenshots. The goal of the lab study was to answer the following research questions based on programmers' experience with real tools that support refactoring in the two compositional and wizard-based paradigms.

- Which paradigm of refactoring do programmers prefer?
- Which paradigm is faster?
- Which paradigm is less error-prone?

6.1 Tool

We implemented an Eclipse plug-in to support Extract Superclass, Extract Interface, Pull Up, and Push Down in the compositional paradigm. Based on the survey study, we improved the design of our tool in several ways. First, we replaced the "Create new superclass in file" action by "Create a new superclass for T in a new file". We made this change to adhere to the coding conventions of Java more strictly. Second, we added an action to the menu called "Create New Superclass" to support multi-selections. When the user selects multiple classes, e.g., in the Package Explorer view, this action would create an empty superclass for the selected classes. Finally, we implemented additional actions such as "Move type T to a new file", where T is a type name, and "Add parameter to method m for expression", where m is a method name (Introduce Parameter in Quick Assist). However, the participants did not use these three actions as they were not applicable to the refactoring task of the lab study.

6.2 Participants

All of our participants were experienced programmers who used Eclipse for Java development at a large software company. We first ran a pilot study on three programmers. Then, we conducted the main study on 13 programmers. Of all the participants, two reported having five to ten years of programming experience and 11 reported more than ten years. One participant reported that he was familiar with Java, nine participants considered themselves as being very familiar with Java, and three indicated that they were experts. One participant reported that he was somewhat familiar with Eclipse, four participants rated themselves as familiar, seven said they were very familiar, and one rated himself as expert.

6.3 Study Design

We instructed each participant to finish the task in both compositional and wizard-based paradigms (*within-subject*). Each participant tried the paradigms in a random order (*counterbalancing*) to overcome the potential *carryover effect*. We did not ask each participant to try only one paradigm (*between-subject*) for several reasons. First, *individual differences* would affect the results of such a study. Second, a between-subject study requires more participants to draw meaningful conclusions. Third, such a study would allow only a quantitative comparison (e.g., efficiency and correctness) of the two paradigms. However, we felt that such measures were not enough to reliably compare the two paradigms. A participant can offer a qualitative comparison only if she tries both paradigms.

At the beginning of the study, we asked the participants to complete a *prequestionnaire*. Then, we asked them to perform some introductory tasks to familiarize themselves with the code. We then instructed each participant to perform the task twice in a *random order*, once using the compositional paradigm and another time using the wizard-based paradigm. Finally, we asked the participants to rate the two paradigms of refactoring in a *postquestionnaire* and participate in a semi-structured interview with us. The study took about an hour for each participant, and we offered a $25 gift card to each participant.

Refactoring Task. We used a refactoring that had occurred in the open-source project `HTMLParser` as our refactoring task. Kerievsky used this refactoring as an example of Extract Composite [11, p. 214] in his book. Several classes of an old revision of the code base exhibit code duplication. These classes contain a list and a method that iterates over the list and computes its string representation. The fields had different names in different classes and the methods accessed the elements in slightly different ways. We asked our participants to remove this code duplication between two classes by extracting the common field and method into a new common superclass. We limited the refactoring to two classes to make it feasible to finish the refactoring in about 20 minutes.

Pilot Study. During the pilot study, we noticed that some participants accidentally introduced subtle errors while refactoring the code. So, we asked the

participants of the main study to check that the unit tests passed at the beginning and end of the study. In addition, we instructed the participants to ensure that the new superclass is only referenced by its subclasses. We decided that the existing uses of the subclasses should not be replaced by the new superclass, because of the existing dynamic type checks and casts.

6.4 Data Analysis

We measured the *task completion time* and checked the *correctness* of the performed task to compare the two paradigms quantitatively. If a participant finished the refactoring task, we compared his resulting code with the expected code in the instructions. If the participant missed some expected changes or introduced unexpected ones, we considered it an incorrect refactoring.

Similar to the analysis of survey comments (Section 5.2), we employed *thematic coding* [32] to *systematically* analyze the retrospective interviews.

6.5 Results

Task Completion Time. The medians of the task completion times using the wizard-based and compositional refactorings were 16.5 and 10.5 minutes, respectively. A Wilcoxon signed-rank test shows that there is a significant effect of refactoring tool on the task completion time ($W = 41$, $Z = 2.25$, $p = 0.02 < 0.05$, $r = 0.50$, two-tailed). Two participants did not finish the refactoring task using either tools during the allotted time. Another participant could not finish the task using the wizards. We excluded these three participants from our significance test. One participant finished the refactoring faster using the wizards. The other nine participants finished the task faster using the compositional paradigm.

Correctness. Participants were more likely to complete the task correctly in the compositional paradigm. Seven participants introduced accidental changes to the code base using the wizard-based refactorings, while only one participant left the refactoring incomplete using the compositional paradigm.

The Extract Superclass refactoring wizard has an option called "Use the extracted class where possible", which is checked by default. This option causes the tool to replace all occurrences of the selected classes by the superclass whenever this replacement does not introduce any compilation problems. The Pull Up refactoring wizard has a similar option. Only three participants unchecked this option on the wizard and only one participant noticed the unexpected changes in the preview and deselected them. The other seven participants were surprised when they discovered unexpected references to the new superclass at the end. At that point, it was difficult to revert the unexpected changes because the participants had already changed the code too much since the application of the wizard-based refactoring. Two participants failed to finish the task using wizards because reverting the unwanted changes was too time-consuming for them.

Qualitative. The majority of our participants were more satisfied with the compositional paradigm, found it easier to learn and use, felt more control and confidence over the refactoring, and expected more opportunities for using it in their code (Table 3).

Table 3. Number of participants of the lab study who preferred each paradigm of refactoring (the first two rows) with respect to each quality (columns). The last row lists the number of participants with no preference.

	control	correctness	ease of learning	ease of use	opportunity to use	satisfaction
compositional	9	6	7	7	7	9
wizard-based	3	5	4	2	1	3
no preference	1	2	2	4	5	1

During the interviews, we asked about the advantages and disadvantages of the two paradigms of refactoring. The following presents the themes that we extracted from the participants' responses.

Control Participants felt they had more control in the compositional paradigm, because the steps are small, predictable, and mimic their manual refactorings. One participant said:

> The wizard gives this illusion of just doing everything for you. [...] The downside is that there were a number of options that I read and didn't quite make sense of, and said I guess I don't have to care about that. And, of course, I found my sorrow that that wasn't true. It did things that I completely didn't expect. [...] And, it doesn't give control.

On the other hand, one participant attributed his feeling of control in the wizard-based paradigm to the previews.

Correctness A participant said:

> The thing that I like about it [compositional paradigm] is that you're taking actions yourself. So, when you see an error, you usually have an idea of which action that you took caused the error.

Another participant explained why he did not trust the correctness of compositional refactorings as much as the wizard-based ones as follows:

> I was not sure if it [the compositional refactoring tool] was seeing the full picture of the changes. Since it was stepwise [and] I'm doing [each step] one by one, I'm not sure if each of the steps is going to be integrated correctly.

In practice, the participants were more likely to refactor incorrectly using wizards.

Change Review. Seven participants preferred the tool to inform them about the effects of an automated refactoring on their code. However, most participants did not inspect the previews of wizard-based refactorings. As a result, seven participants did not catch unintended changes of the wizard-based refactorings in their previews. Four participants mentioned that the previews were too cluttered. One participant said that previews are good for beginners who want to learn a new refactoring wizard. Our prior study showed that programmers rarely preview their refactorings in practice [27].

Multi-selections. Four liked the wizard's ability to refactor multiple entities at the same time. During the study, five participants tried to use the Extract Superclass wizard to extract a common superclass from two classes at once. The rest either found it easier to refactor in smaller steps or did not notice the configuration option to extract from multiple classes.

Configuration Options and Error Messages. Although the wizard-based refactorings provide many options to customize the refactorings, the participants only used a subset of these options. Because one of the methods that the participants had to move to the superclass referenced other members of the subclass, the refactoring wizard reported an error message. The refactoring wizard has an "Add Required" button that when pressed selects all members that are referenced by the currently selected members. However, none of the participants used this configuration option. Instead, they performed the refactoring and fixed the resulting compilation problem manually. One of the participants said that he ignored the error message of the wizard because it was not *actionable*:

> It [The refactoring wizard] came up with something like: "Sorry, this method is referring to this other variable that we can't change". I didn't know what I could do about that in the window. I was like: "OK. Thanks for the information!" [*laughter*]

These observations are consistent with the results of prior studies that showed programmers rarely configure the refactoring wizards [19] and usually apply an automated refactoring that has reported problems [27].

Composition Order. Three participants mentioned that one has to be careful with the order in which she composes the refactorings. On the other hand, one participant indicated that sometimes significant work is required to transform the code to a state that is amenable to the application of a wizard-based refactoring.

6.6 Design Suggestions

The participants suggested improvements to the compositional and wizard-based paradigms. For the compositional paradigm, two participants proposed that the tool suggests the entities that the programmer might want to refactor next.

For the wizard-based paradigm, two participants suggested the ability to match up similar entities. For instance, the Extract Superclass refactoring could

detect similar members in multiple classes, or let the programmer match up the related members and pull them up to the superclass in one step. In addition, one participant proposed that the refactoring wizard provides an *incremental preview*. An incremental preview gets updated as the programmer manipulates the configuration options. Finally, one participant suggested that the tool presents the previews graphically.

7 Limitations

Like any study, each of our prior field study [27], inference of refactoring sets (Section 2) and composition patterns (Section 3), survey (Section 5) and lab study (Section 6) has its own limitations. However, their results with respect to the effectiveness of compositional refactoring corroborate one another. The rest of this section discusses some of the limitations of our work.

Eclipse Foundation Data Set. The Eclipse foundation data (Section 2.1), while huge, lacks precision. For instance, it does not differentiate certain refactorings, e.g., Inline Local Variable and Inline Method. In addition, this data set does not include the project and workspace in which the refactoring is invoked. Moreover, it misses refactorings invoked through Quick Assist. Despite these limitations, the Eclipse foundation data serves as a good starting point to quantify the prevalence of frequent refactoring sets (Section 2).

Participants. The Illinois data set, while more precise, comes from a smaller pool of participants. We found the recruitment challenging due to issues such as privacy, confidentiality, and lack of trust in the reliability of research tools. Nonetheless, our participants come from diverse backgrounds, have various levels of experiences, and work on a variety of nontrivial projects. Thus, we believe that our participants are representative of real-world programmers.

The majority of the lab study participants were very familiar with Eclipse. Further studies are needed to understand the effect of experience on the preferred paradigm of refactoring.

Generalizability. Due the constraints of the survey and lab studies, we evaluated the compositional paradigm using two refactorings, i.e., Extract Superclass and Extract Composite. We have implemented refactorings such as Pull Up, Push Down, and Extract Interface in the compositional paradigm. More evaluation is left to future work.

Our data sets were limited to the use of the Eclipse refactoring tool for Java. However, we expect our results to hold for similar refactoring tools, because they follow a similar user interaction model, i.e., wizard-based refactoring.

Refactoring Tasks. The comments of survey participants were based on screenshots not use of the tools. However, these refactorings are based on familiar features of Eclipse, i.e., wizards and Quick Assist. The insightful comments of the participants indicate that they understood the two paradigms well.

The survey study relied on a small piece of code. We intentionally kept the survey simple to make it understandable for programmers who may not be familiar with the intricacies of the refactoring wizards.

For the lab study, we selected a single realistic refactoring task that Kerievsky used in his book to introduce Extract Composite [11, p. 214]. As some of our participants speculated, the wizard-based refactoring might be more appropriate for refactorings that affect hundreds of files. Further studies are needed to compare the two paradigms of refactoring on a variety of refactoring tasks.

Participant Response Bias. A common limitation of user studies is that participants may favor the interface that they think the researcher has developed. However, we think that our results are less affected by this bias, because most of our participants could not tell which interface was ours. At the end of the lab study, most participants asked us which interface was ours. This is because the Extract Superclass wizard is rarely used [19,27], and few programmers remember all Quick Assist actions to identify the actions contributed by our plug-in.

8 Related Work

Composite Refactorings. One way of automating composite refactorings is to build tools that execute a sequence of smaller refactorings atomically [16,30]. We introduce a radically different paradigm. Rather than building a monolithic tool from several refactorings, we propose that a large refactoring be decomposed into smaller ones. These two paradigms suit different needs. The monolithic paradigm is suited for toolsmiths who are in charge of applying a refactoring on a large code base in batch mode. The compositional paradigm is designed for interactive refactoring in an IDE. In addition, the monolithic paradigm aims to provide correctness guarantees by inferring preconditions [3, 14, 24]. The compositional paradigm makes it easy for the programmers to verify the correctness of the refactoring by making each step easy to predict and verify.

Murphy-Hill et al. [19] reported that developers often repeat an automated refactoring within 60 seconds. Negara et al. [20] found that more than one third of manual and automated refactorings are performed in batches. Our study goes beyond reporting the frequencies of refactoring sets (Section 2) and sheds light on the rationales of composing automated refactorings (Section 3).

Schäfer et al. [26] argued that a very fine-grained decomposition of a refactoring into a composition of micro-refactorings over an extended language makes the implementation of the refactoring tool more reliable. They used Extract Method as an example to demonstrate their technique. While their focus was on reliability, ours is on usability. Generalizing their results, the compositional paradigm should lead to more reliable implementations of large refactorings.

Usability of Refactoring Tools. Empirical studies [19,27] showed that refactoring tools are underused. These results prompted research on improving the usability of refactoring tools.

Murphy-Hill and Black [18] developed a prototype tool that visualizes code selections and error messages. They evaluated their tool on a single refactoring, namely Extract Method. Similarly, we evaluated compositional refactoring on two refactorings, Extract Superclass and Extract Composite.

Researchers have proposed refactoring auto-completion systems [5,8], which prompt programmers to automatically complete a manual refactoring. Others have proposed alternative methods of invoking refactorings drag-and-drop [15] and multi-touch [22] gestures. While these systems aim to make the *invocation* of refactoring tools seamless, compositional refactoring makes automated refactoring more *predictable*. Nonetheless, alternative methods of invocations are complementary to our work, because they can streamline the invocation of the individual steps of a composite refactoring.

9 Future Work

We evaluated the compositional paradigm using a survey and a lab study. Future research can evaluate this paradigm in the field for more refactorings and program transformations in general.

One area of future work would be to investigate other ways of assisting programmers in composing refactorings. For example, how accurately can a tool predict the next refactoring that a programmer may invoke in a compositional paradigm? Can a history of previously invoked refactorings and frequent refactoring sets be used to accurately make such a prediction?

Another line of future work would be to study the pedagogical aspects of the compositional paradigm. While our discussion of refactoring composition patterns (Section 3) could serve as a starting point for learning these patterns, more research is needed to deliver a more comprehensive catalog. One technique to make people adopt new skills is to make it easy to learn from their peers. How can we facilitate the transfer of refactoring composition skills in a team?

10 Conclusions

We feel a rush to more automation in the software engineering community, often through wishful thinking or superficial claims about the impact of additional automation on the productivity of programmers. Despite the push to automate more refactorings and other recurring program transformations, studies have shown that programmers greatly underuse such tools [18,20,27].

Rather than offering more automation, we took the opposite direction, and proposed the *compositional paradigm* for refactoring. In this paradigm, the tool automates the individual steps, and puts the programmer in control by letting her manually compose the steps into a complex change.

The compositional paradigm was inspired by our analysis of the refactoring practices of programmers in the wild. Our data mining and manual examination of two refactoring usage data sets provided evidence for the *prevalence*, *diversity*, *rationales*, and *naturalness* of composing automated refactorings. In addition, our survey and lab studies showed that the compositional paradigm is more *effective* than the existing wizard-based paradigm of refactoring.

The compositional paradigm outperforms the wizard-based one by reducing the automation level. Although this result may seem counterintuitive, it is not unique to software engineering. Designers of other fields, e.g., aviation, healthcare, and manufacturing, struggle with similar problems. What is an appropriate level of automation? What should the role of the human operator be? Often, researchers find that *less is more*. That is, a modest design, which provides clear, immediate feedback, outperforms a design with a high level of automation that does not integrate the human operator well [2, 12, 31].

Acknowledgment. We thank Deepak Azad, Brian Bailey, Wayne Beaton, Kent Beck, Tracy Bialik, Robert Bowdidge, John Brant, Danny Dig, Kyle Doren, Darko Marinov, Yalan Meng, David Morgenthaler, Chris Parnin, Russ Ruffer, Marjan Sirjani, the attendees of the software engineering seminar at Illinois, and the programmers who participated in our studies. This work was partially supported by the National Science Foundation grant number CCF 11-17960 and the United States Department of Energy under contract number DE-F02-06ER25752.

References

1. The R Project for Statistical Computing, `http://www.r-project.org/`
2. Bainbridge, L.: Ironies of Automation. Automatica, 775–779 (1983)
3. Cinnéide, M.Ó.: Automated Application of Design Patterns: A Refactoring Approach. Ph.D. thesis, Univ. of Dublin, Trinity College (2000)
4. Dig, D., Marrero, J., Ernst, M.D.: Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In: Proc. ICSE, pp. 397–407 (2009)
5. Foster, S., Griswold, W.G., Lerner, S.: WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In: Proc. ICSE, pp. 222–232 (2012)
6. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
8. Ge, X., DuBose, Q.L., Murphy-Hill, E.: Reconciling Manual and Automatic Refactoring. In: Proc. ICSE, pp. 211–221 (2012)
9. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. Ph.D. thesis, Univ. of Washington (1991)
10. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques, 3rd edn. Morgan Kaufmann (2011)
11. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004)
12. Kirlik, A.: Modeling Strategic Behavior in Human-Automation Interaction: Why an "Aid" Can (and Should) Go Unused. J. Human Factors and Ergonomics Soc., 221–242 (1993)

13. Kjolstad, F., Dig, D., Acevedo, G., Snir, M.: Transformation for Class Immutability. In: Proc. ICSE. pp. 61–70 (2011)
14. Kniesel, G., Koch, H.: Static Composition of Refactorings. Sci. Comput. Program. pp. 9–51 (2004)
15. Lee, Y.Y., Chen, N., Johnson, R.E.: Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In: ICSE (2013)
16. Li, H., Thompson, S.: A Domain-Specific Language for Scripting Refactorings in Erlang. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 501–515. Springer, Heidelberg (2012)
17. Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE? IEEE Software pp. 76–83 (2006)
18. Murphy-Hill, E., Black, A.P.: Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In: Proc. ICSE, pp. 421–430 (2008)
19. Murphy-Hill, E., Parnin, C., Black, A.P.: How We Refactor, and How We Know It. IEEE Trans. Software Eng. pp. 5–18 (2011)
20. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A Comparative Study of Manual and Automated Refactorings. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 552–576. Springer, Heidelberg (2013)
21. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, Univ. of Illinois at Urbana-Champaign (1992)
22. Parnin, C., Görg, C., Rugaber, S.: CodePad: Interactive Spaces for Maintaining Concentration in Programming Environments. In: Proc. SoftVis. pp. 15–24 (2010)
23. Roberts, D., Brant, J., Johnson, R.: A Refactoring Tool for Smalltalk. Theor. Pract. Object Syst. pp. 253–263 (1997)
24. Roberts, D.B.: Practical Analysis for Refactoring. Ph.D. thesis, Univ. of Illinois at Urbana-Champaign (1999)
25. Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Refactoring Java Programs for Flexible Locking. In: Proc. ICSE. pp. 71–80 (2011)
26. 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)
27. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, Disuse, and Misuse of Automated Refactorings. In: Proc. ICSE, pp. 233–243 (2012)
28. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Zilouchian Moghaddam, R., Johnson, R.E.: The Need for Richer Refactoring Usage Data. In: Proc. PLATEAU, pp. 31–38 (2011)
29. Vakilian, M., Dig, D., Bocchino Jr., R.L., Overbey, J.L., Adve, V., Johnson, R.: Inferring Method Effect Summaries for Nested Heap Regions. In: Proc. ASE, pp. 421–432 (2009)
30. Verbaere, M., Ettinger, R., de Moor, O.: JunGL: A Scripting Language for Refactoring. In: Proc. ICSE, pp. 172–181 (2006)
31. Vicente, K.J.: Less is (sometimes) more in cognitive engineering: the role of automation technology in improving patient safety. Qual Saf Health Care, pp. 291–294 (2003)
32. Ryan, W., Bernard, G.,, H.R.: Data Management and Analysis Methods. In: Handbook of Qualitative Research, 2nd edn. SAGE Publications (2011)
33. Wloka, J., Sridharan, M., Tip, F.: Refactoring for Reentrancy. In: Proc. ESEC/FSE, pp. 173–182 (2009)
34. Xing, Z., Stroulia, E.: Refactoring Practice: How it is and How it Should be Supported – An Eclipse Case Study. In: Proc. ICSM, pp. 458–468 (2006)

A Comparative Study of Manual and Automated Refactorings

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

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

Abstract. Despite the enormous success that manual and automated refactoring has enjoyed during the last decade, we know little about the practice of refactoring. Understanding the refactoring practice is important for developers, refactoring tool builders, and researchers. Many previous approaches to study refactorings are based on comparing code snapshots, which is *imprecise*, *incomplete*, and does not allow answering research questions that involve time or compare manual and automated refactoring.

We present the first extended empirical study that considers both manual and automated refactoring. This study is enabled by our algorithm, which infers refactorings from *continuous* changes. We implemented and applied this algorithm to the code evolution data collected from 23 developers working in their natural environment for 1,520 hours. Using a corpus of 5,371 refactorings, we reveal several new facts about manual and automated refactorings. For example, more than half of the refactorings were performed manually. The popularity of automated and manual refactorings differs. More than one third of the refactorings performed by developers are clustered in time. On average, 30% of the performed refactorings do not reach the Version Control System.

1 Introduction

Refactoring [10] is an important part of software development. Development processes like eXtreme Programming [3] treat refactoring as a key practice. Refactoring has revolutionized how programmers design software: it has enabled programmers to continuously explore the design space of large codebases, while preserving the existing behavior. Modern IDEs such as Eclipse, NetBeans, IntelliJ IDEA, or Visual Studio incorporate refactoring in their top menu and often compete on the basis of refactoring support.

Several research projects [7, 17, 18, 23–25, 27, 31, 33] made strides into understanding the practice of refactoring. This is important for developers, refactoring tool builders, and researchers. Tool builders can improve the current generation of tools or design new tools to match the practice, which will help developers to

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 552–576, 2013.

perform their daily tasks more effectively. Understanding the practice also helps researchers by validating or refuting assumptions that were previously based on folklore. It can also focus the research attention on the refactorings that are popular in practice. Last, it can open new directions of research. For example, in this study we discovered that more than one third of the refactorings performed in practice are applied in a close time proximity to each other, thus forming a *cluster*. This result motivates new research into refactoring composition.

The fundamental technical problem in understanding the practice is being able to identify the refactorings that were applied by developers. There are a few approaches. One is to bring developers in the lab and watch how they refactor [24]. This has the advantage of observing all code changes, so it is precise. But this approach studies the programmers in a confined environment, for a short period of time, and thus, it is *unrepresentative*.

Another approach is to study the refactorings applied in the wild. The most common way is to analyze two Version Control System (VCS) snapshots of the code either manually [2, 7, 21, 22] or automatically [1, 4, 6, 15, 19, 29, 32]. However, the snapshot-based analysis has several disadvantages. First, it is *imprecise*. Many times refactorings overlap with editing sessions, e.g., a method is both renamed, and its method body is changed dramatically. Refactorings can also overlap with other refactorings, e.g., a method is both renamed and its arguments are reordered. The more overlap, the more noise. Our recent study [27] shows that 46% of refactored program entities are also edited or further refactored in the same commit. Second, it is *incomplete*. For example, if a method is renamed more than once, a snapshot-based analysis would only infer the last refactoring. Third, it is *impossible* to answer many empirical questions. For example, from snapshots we cannot determine how long it takes developers to refactor, and we cannot compare manual vs. automated refactorings.

Others [25, 31] have studied the practice of automated refactorings recorded by Eclipse [7, 16], but this approach does not take into account the refactorings that are applied manually. Recent studies [24, 25, 31] have shown that programmers sometimes perform a refactoring manually, even when the IDE provides an automated refactoring. Thus, this approach is *insufficient*.

We present the first empirical study that addresses these five serious limitations. We study the refactoring practice in the wild, while employing a *continuous* analysis. Such analysis tracks code changes as soon as they happen rather than inferring them from VCS snapshots. We study synergistically the practice of both manual and automated refactorings. We answer seven research questions:

RQ1: What Is the Proportion of Manual vs. Automated Refactorings?
RQ2: What Are the Most Popular Automated and Manual Refactorings?
RQ3: How Often Does a Developer Perform Manual vs. Automated Refactorings?
RQ4: How Much Time Do Developers Spend on Manual vs. Automated Refactorings?
RQ5: What is the Size of Manual vs. Automated Refactorings?
RQ6: How Many Refactorings Are Clustered?

RQ7: How Many Refactorings Do Not Reach VCS?

Answering these empirical questions requires us to infer refactorings from continuous code changes. Recent tools [9,14] that were developed for such inference were not designed for empirical studies. Therefore, we designed and implemented our own refactoring inference algorithm that analyzes code changes continuously. Currently, our algorithm infers ten kinds of refactorings performed either manually or automatically, but it can be easily extended to handle other refactorings as well. The inferred ten kinds of refactorings were previously reported [31] as the most popular among automated refactorings. Table 1 shows the inferred refactorings, ranging from API-level refactorings (e.g., Rename Class), to partially local (e.g., Extract Method), to completely local refactorings (e.g., Extract Local Variable). We think the inferred refactorings are representative since they are both popular and cover a wide range of common refactorings that operate on different scope levels. In the following, when we refer to refactorings we mean these ten refactoring kinds.

Table 1. Inferred refactorings. *API-level* refactorings operate on the elements of a program's API. *Partially local* refactorings operate on the elements of a method's body, but also affect the program's API. *Completely local* refactorings affect elements in the body of a single method only.

Scope	Refactoring
API-level	Encapsulate Field
	Rename Class
	Rename Field
	Rename Method
Partially local	Convert Local Variable to Field
	Extract Constant
	Extract Method
Completely local	Extract Local Variable
	Inline Local Variable
	Rename Local Variable

In our previous study [27], we continuously inferred Abstract Syntax Tree (AST) node operations, i.e., *add*, *delete*, and *update* AST node from fine-grained code edits (e.g., typing characters). In this study, we designed and implemented an algorithm that infers refactorings from these AST node operations. First, our algorithm infers high-level properties, e.g., replacing a variable reference with an expression. Then, from combination of properties it infers refactorings. For example, it infers that a local variable was inlined when it noticed that a variable declaration is deleted, and all its references are replaced with the initialization expression.

We applied our inference algorithm on the real code evolution data from 23 developers, working in their natural environment for 1,520 hours. We found that more than half of the refactorings were performed manually, and thus,

the existing studies that focus on automated refactorings only might not be generalizable since they consider less than half of the total picture. We also found that the popularity of automated and manual refactorings differs. Our results present a fuller picture about the popularity of refactorings in general, which should help both researchers and tool builders to prioritize their work. Our findings provide an additional evidence that developers underuse automated refactoring tools, which raises the concern of the usability problems in these tools. We discovered that more than one third of the refactorings performed by developers are clustered. This result emphasizes the importance of researching refactoring clusters in order to identify refactoring composition patterns. Finally, we found that 30% of the performed refactorings do not reach the VCS. Thus, using VCS snapshots alone to analyze refactorings might produce misleading results.

This paper makes the following contributions:

1. We answered seven research questions about the practice of manual and automated refactoring and discovered several new facts.
2. We designed and implemented an algorithm that employs continuous change analysis to infer refactorings. Our inference algorithm and infrastructure have been successfully evaluated by the ECOOP artifact evaluation committee and found to meet expectations. Our implementation is open source and available at http://codingtracker.web.engr.illinois.edu.
3. We evaluated our algorithm on a large corpus of *real world* data.

2 Research Methodology

To answer our research questions, we employed the code evolution data that we collected as part of our previous user study [27] on 23 participants. We recruited 10 professional programmers who worked on different projects in domains such as marketing, banking, business process management, and database management. We also 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.

The participants of our study have different affiliations, programming experience, and used our tool, CodingTracker [27], for different amounts of time. Consequently, the total *aggregated* data is non-homogeneous. To see whether this non-homogeneity affects our results, we divided our participants into seven groups along the three above mentioned categories. Table 2 shows the detailed statistics for each group as well as for the aggregated data. For every research question, we first present the aggregated result, and then discuss any discrepancies between the aggregated and the group results.

To collect code evolution data, we asked each participant to install the 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 and

Table 2. Size and usage time statistics of the aggregated and individual groups

Metric	Group							
	Aggregated	Affiliation		Tool usage (hours)		Programming experience (years)		
		Students	Professionals	≤ 50	> 50	< 5	$5 - 10$	> 10
Participants	23	13	10	13	10	5	11	6
Usage time, hours	1,520	1,048	471	367	1,152	269	775	458
Mean	66	81	47	28	115	54	70	76
STDEV	52	54	44	16	38	46	52	62

interactions with Version Control System (VCS). CodingTracker employed existing infrastructure [31] to regularly upload the collected data to our centralized repository.

At the time when CodingTracker recorded the data, we did not have a refactoring inference algorithm. However, CodingTracker can accurately replay all the code editing events, thus recreating an exact replica of the evolution session that happened in reality. We replayed the coding sessions and this time, we applied our newly developed refactoring inference algorithm.

We first applied our AST node operations inference algorithm [27] on the collected raw data to represent code changes as *add*, *delete*, and *update* operations on the underlying AST. These basic AST node operations serve as input to our refactoring inference algorithm. Section 4 presents more details about our refactoring inference algorithm.

Next, we answer every research question by processing the output of the algorithm with the question-specific analyzer. Note that our analyzers for **RQ1** – **RQ5** ignore *trivial* refactorings. We consider a refactoring trivial if it affects a single line of code, e.g., renaming a variable with no uses.

3 Research Questions

RQ1: What Is the Proportion of Manual vs. Automated Refactorings? Previous research on refactoring practice either predominantly focused on automated refactorings [23, 25, 31] or did not discriminate manual and automated refactorings [7,33]. Answering the question about the relative proportion of manual and automated refactorings will allow us to estimate how *representative* automated refactorings are of the total number of refactorings, and consequently, how general are the conclusions based on studying automated refactorings only.

For each of the ten refactoring kinds inferred by our algorithm, we counted how many refactorings were applied using Eclipse automated refactoring tools and how many of the inferred refactorings were applied manually. Fig. 1 shows our aggregated results. The last column represents the combined result for all the ten refactoring kinds.

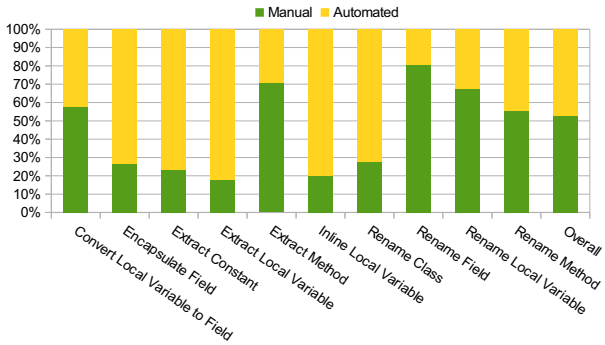


Fig. 1. Relative proportion of manual and automated refactorings

Overall, our participants performed around 11% more manual than automated refactorings (2,820 vs. 2,551). Thus, research focusing on automated refactorings considers less than half of the total picture. Moreover, half of the refactoring kinds that we investigated, Convert Local Variable to Field, Extract Method, Rename Field, Rename Local Variable, and Rename Method, are predominantly performed manually. This observation undermines the generalizability of the existing studies based on the automated execution of these popular refactorings. Also, it raises concerns for tool builders about the underuse of the automated refactoring tools, which could be a sign that these tools require a considerable improvement.

We compared the number of manual and automated refactorings performed by each group. Table 3 shows the total counts of manual and automated refactorings as well as the relative fraction of manual over automated refactorings. The results for the groups in the **Affiliation** and **Usage time** categories are consistent with the results for the aggregated data. At the same time, the programming experience of our participants has a greater impact on the ratio of the performed manual and automated refactorings. In particular, developers with less than five years of programming experience tend to perform 28% more manual than automated refactorings, while those with an average experience (5 – 10 years) perform more automated than manual refactorings. This result reflects a common intuition that novice developers are less familiar with the refactoring tools (e.g., in **RQ3** we observed that novices do not perform three kinds of automated refactorings at all), but start using them more often as their experience grows. Nevertheless, developers with more than ten years of experience perform many more manual than automated refactorings (49%). One of the reasons for this behavior could be that more experienced developers learned to perform refactorings well before the appearance of the refactoring tools. Also, experts might think that they are faster without the refactoring tool. For example, we observed that such developers mostly perform manually *Rename* refactorings, which could be accomplished quickly (but less reliably) using the *Search & Replace* command.

RQ2: What Are the Most Popular Automated and Manual Refactorings? Prior studies [23, 31] identified the most popular automated refactorings to better understand how developers refactor their code. We provide a

Table 3. Manual and automated refactorings performed by each group

Category	Group	Manual	Automated	Manual over Automated
Aggregated	All data	2820	2551	10.5%
Affiliation	Students	1645	1516	8.5%
	Professionals	1175	1035	13.5%
Usage time	$\leq$ 50 hours	485	471	3%
	$>$ 50 hours	2335	2080	12.3%
Experience	$<$ 5 years	292	228	28%
	5 – 10 years	1282	1459	-12.1%
	$>$ 10 years	1237	829	49.2%

more complete picture of the refactoring popularity by looking at both manual and automated refactorings. Additionally, we would like to contrast how similar or different are popularities of automated refactorings, manual refactorings, and refactorings in general.

To measure the popularity of refactorings, we employ the same refactoring counts that we used to answer the previous research question. Fig. 2, 3, and 4 correspondingly show the popularity of automated, manual, and all refactorings in the aggregated data. The Y axis represents refactoring counts. The X axis shows refactorings ordered from the highest popularity rank at the left to the lowest rank at the right.

Our results on popularity of automated refactorings mostly corroborate previous findings [31][1]. The only exceptions are the Inline Local Variable refactoring, whose popularity has increased from the seventh to the third position, and the Encapsulate Field refactoring, whose popularity has declined from the fifth to the seventh position. Overall, our results show that the popularity of automated and manual refactorings is quite different: the top five most popular automated and manual refactorings have only three refactorings in common — Rename Local Variable, Rename Method, and Extract Local Variable, and even these refactorings have different ranks. The most important observation though is that the popularity of automated refactorings does not reflect well the popularity of refactorings in general. In particular, the top five most popular refactorings and automated refactorings share only three refactorings, out of which only one, Rename Method, has the same rank. Having a fuller picture about the popularity of refactorings, researchers would be able to automate or infer the refactorings that are popular when considering both automated and manual refactorings. Similarly, tool builders should pay more attention to the support of the popular refactorings. Finally, novice developers might decide what refactorings to learn first depending on their relative popularity.

Refactoring popularity among different participant groups is mostly consistent with the one observed in the aggregated data. In particular, for three groups (Usage time $>$ 50 hours, Experience 5 – 10 years, and Experience $>$ 10 years),

[1] Note that we can not directly compare our results with the findings of Murphy et al. [23] since their data represents the related refactoring kinds as a single category (e.g., Rename, Extract, Inline, etc.).

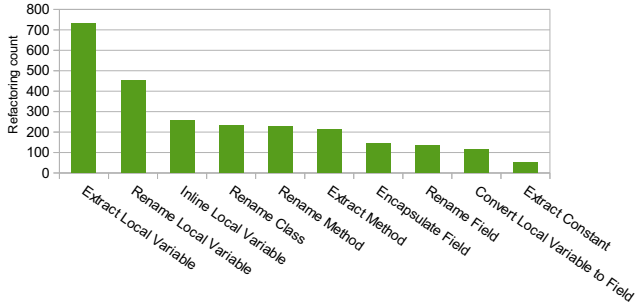


Fig. 2. Popularity of automated refactorings

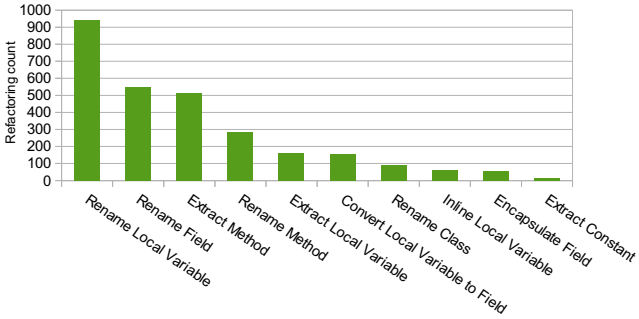


Fig. 3. Popularity of manual refactorings

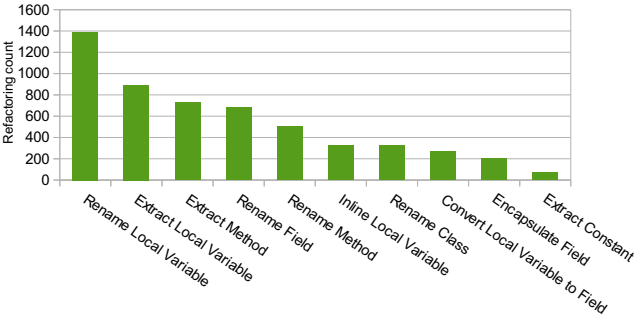


Fig. 4. Popularity of refactorings

the top five most popular refactorings are the same as for the aggregated data, while for the rest of the groups, four out of five most popular refactorings are the same.

RQ3: How Often Does a Developer Perform Manual vs. Automated Refactorings? In our previous study [31], we showed that developers may underuse automated refactoring tools for a variety of reasons, one of the most important being that developers are simply unaware of automated refactoring tools. Answering this question will help us to better understand whether developers who are aware about an automated refactoring tool use the tool rather than refactor manually.

In the following, we denote the quantity of automated tool usage as A. We compute A as a ratio of automated refactorings to the total number of refactorings of a particular kind performed by an individual participant. For each of the ten inferred refactoring kinds, we counted the number of participants for a range of values of A, from $A = 0\%$ (those who never use an automated refactoring tool) up to $A = 100\%$ (those who always use the automated refactoring tool).

Both for the aggregated data and for all groups, we observed that the fraction of participants who always perform a refactoring manually is relatively high for all the ten refactoring kinds (with a few exceptions). Also, in the aggregated data, there are no participants who apply Convert Local Variable to Field, Encapsulate Field, Extract Method, and Rename Field using the automated refactoring tools only. In groups, there are even more refactoring kinds that are never applied using automated refactoring tools only. Overall, our results provide a stronger quantitative support for the previously reported findings [25, 31] that the automated refactoring tools are underused.

To get a better insight into the practice of manual vs. automated refactoring of our participants, we defined three properties:

- **High full automation:** The number of participants who always perform the automated refactoring ($A = 100\%$) is higher than the number of participants who always perform this refactoring manually ($A = 0\%$).
- **High informed underuse:** The number of participants who are aware about the automated refactoring, but still apply it manually most of the time ($0\% < A \leq 50\%$) is higher than the number of participants who apply this refactoring automatically most of the time ($50\% < A \leq 100\%$).
- **General informed underuse:** The number of participants who apply the automated refactoring only ($A = 100\%$) is significantly lower than the number of participants who both apply the automated refactoring and refactor manually ($0\% < A < 100\%$).

Table 4 shows refactoring kinds that satisfy the above properties for each group as well as for the aggregated data. For each group, we present only the difference with the aggregated result, where "−" marks those refactoring kinds that are present in the aggregated result, but are absent in the group result, and "+" is used in the vice-versa scenario.

Our aggregated results show that only for two refactorings, Extract Constant and Rename Class, the number of participants who always perform the

Table 4. Manual vs. automated refactoring practice

Category	Group	Property		
		High full automation	**High informed underuse**	**General informed underuse**
Aggregated		Extract Constant Rename Class	Extract Method Rename Local Variable	All
Affiliation	Students	-Extract Constant -Rename Class	+Conv. Loc. Var. to Field +Extract Constant +Rename Method	
	Professionals	+Rename Method	-Rename Local Variable	-Extract Constant -Rename Class -Rename Method
Usage time	$\leq$ 50 hours	-Extract Constant +Rename Method		-Extract Constant -Rename Class
	$>$ 50 hours	-Rename Class	+Rename Method	-Extract Constant
Experience	$<$ 5 years	-Extract Constant -Rename Class		-Conv. Loc. Var. to Field -Extract Constant -Inline Local Variable
	5 – 10 years	-Extract Constant	-Rename Local Variable +Conv. Loc. Var. to Field +Extract Constant	-Extract Constant
	$>$ 10 years	-Extract Constant	+Encapsulate Field +Rename Method	

automated refactoring is higher than the number of participants who always perform the refactoring manually. Another important observation is that for two refactoring kinds, Extract Method and Rename Local Variable, the number of participants who are aware of the automated refactoring, but still apply it manually most of the time is higher than the number of participants who apply this refactoring automatically most of the time. This shows that some automated refactoring tools are underused even when developers are aware of them and apply them from time to time. Our results for groups show that students tend to underuse more refactoring tools than professionals. Also, developers with more than five years of experience underuse more refactoring tools that they are aware of than those with less than five years of experience. At the same time, novice developers do not use three refactoring tools at all, i.e., they always perform the Convert Local Variable to Field, Extract Constant, and Inline Local Variable refactorings manually. Thus, novice developers might underuse some refactoring tools due to lack of awareness, an issue identified in a previous study [31].

The aggregated result shows that for each of the ten refactoring kinds, the number of participants who apply the automated refactoring only is significantly lower than the number of participants who both apply the automated refactoring and refactor manually. The result across all groups shows that no less than seven refactoring kinds satisfy this property. These results show that developers underuse automated refactoring tools, some more so than the others, which could be an indication of a varying degree of usability problems in these tools.

RQ4: How Much Time Do Developers Spend on Manual vs. Automated Refactorings? One of the major arguments in favor of performing a refactoring automatically is that it takes less time than performing this refactoring manually [30]. We would like to assess this time difference as well as compare the average durations of different kinds of refactorings performed manually.

To measure the duration of a manual refactoring, we consider all AST node operations that contribute to it. Our algorithm marks AST node operations that contribute to a particular inferred refactoring with a generated refactoring's ID, which allows us to track each refactoring individually. Note that a developer might intersperse a refactoring with other code changes, e.g., another refactoring, small bug fixes, etc. Therefore, to compute the duration of a manual refactoring, we cannot subtract the timestamp of the first AST node operation that contributes to it from the timestamp of the last contributing AST node operation. Instead, we compute the duration of each contributing AST node operation separately by subtracting the timestamp of the preceding AST node operation (regardless of whether it contributes to the same refactoring or not) from the timestamp of the contributing AST node operation. If the obtained duration is greater than two minutes, we discard it, since it might indicate an interruption in code editing, e.g., a developer might get distracted by a phone call or take a break. Finally, we sum up all the durations of contributing AST node operations to obtain the duration of the corresponding refactoring.

We get the durations of automated refactorings from CodingSpectator [31]. CodingSpectator measures configuration time of a refactoring performed automatically, which is the time that a developer spends in the refactoring's dialog box. Note that the measured time includes neither the time that the developer might need to check the correctness of the performed automated refactoring nor the time that it takes Eclipse to actually change the code, which could range from a couple of milliseconds to several seconds, depending on the performed refactoring kind and the underlying code.

Fig. 5 shows our aggregated results. On average, manual refactorings take longer than their automated counterparts with a high statistical significance ($p < 0.0001$, using two-sided unpaired t-test) only for Extract Local Variable, Extract Method, Inline Local Variable, and Rename Class since for the other refactoring kinds our participants rarely used the configuration dialog boxes. This observation is also statistically significant across all groups. Manual execution of the Convert Local Variable to Field refactoring takes longer than the automated one with a sufficient statistical significance ($p < 0.04$) for the aggregated data, while for most groups this observation is not statistically significant. The most time consuming, both manually and automatically, is the Extract Method refactoring, which probably could be explained by its complexity and the high amount of code changes involved. All other refactorings are performed manually on average in under 15 – 25 seconds. Some refactorings take longer than others. A developer could take into account this difference when deciding what automated refactoring tool to learn first.

Another observation is that the Rename Field refactoring is on average the fastest manual refactoring. It takes less time than the arguably simpler Rename Local Variable refactoring. One of the possible explanations is that developers perform the Rename Field refactoring manually when it does not require many changes, e.g., when there are few references to the renamed field, which is supported by our results for the following question.

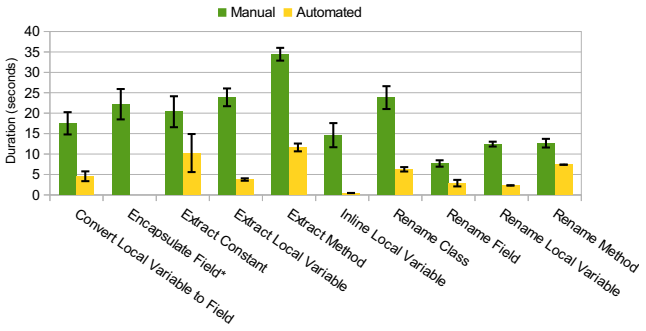


Fig. 5. Average duration of performing manual refactorings and configuring automated refactorings. The black intervals represent the standard error of the mean (SEM). The configuration time bar for the Encapsulate Field refactoring is missing since we do not have data for it.

RQ5: What is the Size of Manual vs. Automated Refactorings? In an earlier project [31], we noticed that developers mostly apply automated refactoring tools for small code changes. Therefore, we would like to compare the average size of manual and automated refactorings to better understand this behavior of developers.

To perform the comparison, we measured the size of manual and automated refactorings as the number of the affected AST nodes. For manual refactorings, we counted the number of AST node operations contributing to a particular refactoring. For automated refactorings, we counted all AST node operations that appear in between the start and the finish refactoring operations recorded by CodingTracker. Note that all operations in between the start and the finish refactoring operations represent the effects of the corresponding automated refactoring on the underlying code [27].

Fig. 6 shows our aggregated results. On average, automated refactorings affect more AST nodes than manual refactorings for four refactoring kinds, Convert Local Variable to Field, Extract Method, Rename Field, and Rename Local Variable, with a high statistical significance ($p < 0.0001$), and for three refactoring kinds, Extract Local Variable, Inline Local Variable, and Rename Method, with a sufficient statistical significance ($p < 0.03$). One of the reasons could be that developers tend to perform smaller refactorings manually since such refactorings have a smaller overhead. At the same time, this observation is not statistically significant for all the above seven refactoring kinds in every group. In particular, it is statistically significant in five out of seven groups for three refactoring kinds, Convert Local Variable to Field, Extract Method, and Rename Field, and in fewer groups for the other four kinds of refactorings.

Intuitively, one could think that developers perform small refactorings by hand and large refactorings with a tool. On the contrary, our findings show that developers perform manually even large refactorings. In particular, Extract Method is by far the largest refactoring performed both manually and automatically – it

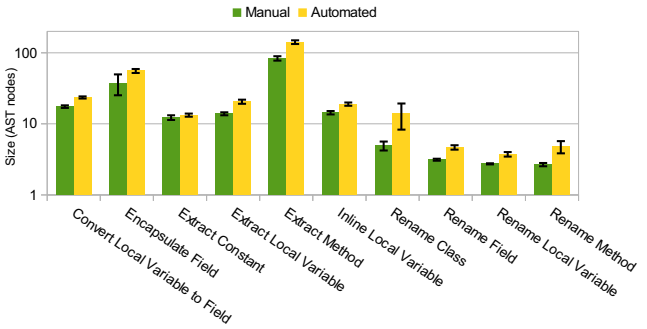


Fig. 6. Average size of manual and automated refactorings expressed as the number of the affected AST nodes. The black intervals represent the standard error of the mean (SEM). The scale of the Y axis is logarithmic.

is more than two times larger than Encapsulate Field, which is the next largest refactoring. At the same time, according to our result for **RQ3**, most of the developers predominantly perform the Extract Method refactoring manually in spite of the significant amount of the required code changes. Thus, the size of a refactoring is not a decisive factor for choosing whether to perform it manually or with a tool. This also serves as an additional indication that the developers might not be satisfied with the existing automation of the Extract Method refactoring [24].

RQ6: How Many Refactorings Are Clustered? To better understand and support refactoring activities of developers, Murphy-Hill et al. [25] identified different refactoring patterns, in particular, *root canal* and *floss* refactorings. A root canal refactoring represents a consecutive sequence of refactorings that are performed as a separate task. Floss refactorings, on the contrary, are interspersed with other coding activities of a developer. In general, grouping several refactorings in a single cluster might be a sign of a higher level refactoring pattern, and thus, it is important to know how many refactorings belong to such clusters.

To detect whether several refactorings belong to the same cluster, we compute a ratio of the number of AST node operations that are part of these refactorings to the number of AST node operations that happen in the same time window as these refactorings, but do not belong to them (such operations could happen either in between refactorings or could be interspersed with them). If this ratio is higher than a particular threshold, T, we consider that the refactorings belong to the same cluster. That is, rather than using a specific time window, we try to get as large clusters as possible, adding refactorings to a cluster as long as the ratio of refactoring to non-refactoring changes in the cluster does not fall below a particular threshold. The minimum size of a cluster is three. Note that for the clustering analysis we consider automated refactorings of all kinds and manual refactorings of the ten kinds inferred by our tool.

Fig. 7 shows the proportion of clustered and separate refactorings for aggregated data for different values of T, which we vary from 1 to 10. $T = 1$ means that the amount of non-refactoring changes does not exceed the amount of refactoring changes in the same cluster. Fig. 8 shows the average size of gaps between separate

refactorings (i.e., refactorings that do not belong to any cluster) expressed as the number of AST node operations that happen in between two separate refactorings or a separate refactoring and a cluster.

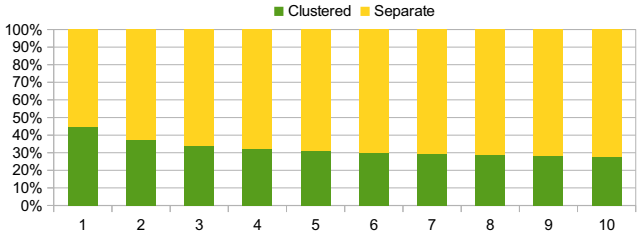


Fig. 7. Proportion of clustered and separate refactorings for different values of the threshold T

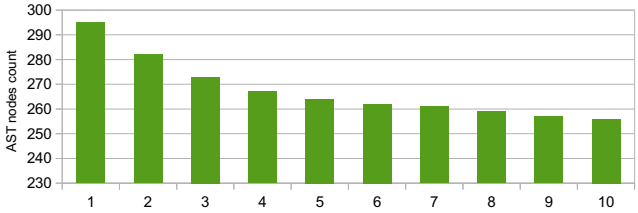


Fig. 8. The average size of gaps between separate refactorings expressed as the number of AST node operations. The X axis represents the values of the threshold T.

Our aggregated results show that for $T = 1$, 45% of the refactorings are clustered. When the threshold grows, the number of the clustered refactorings goes down, but not much — even for $T = 10$, 28% of refactorings are clustered. The average gap between floss refactorings is not very sensitive to the value of the threshold as well. Overall, developers tend to perform a significant fraction of refactorings in clusters. This observation holds for all groups except for the novice developers, where for $T = 1$ only 8% of the refactorings are clustered. One of the reasons could be that novices tend to refactor sporadically, while more experienced developers perform refactorings in chunks, probably composing them to accomplish high-level program transformations (e.g., refactor to a design pattern). Our results emphasize the importance of researching refactoring clusters in order to identify refactoring composition patterns.

RQ7: How Many Refactorings Do Not Reach VCS? Software evolution researchers [6, 8, 11–13, 20, 34] use file-based Version Control Systems (VCSs), e.g., Git, SVN, CVS, as a convenient way to access the code histories of different applications. In our previous study [27], we showed that VCS snapshots provide incomplete and imprecise evolution data. In particular, we showed that 37% of code changes do not reach VCS. Since refactorings play an important role

in software development, in this study, we would like to assess the amount of refactorings that never make it to VCS, and thus, are missed by any analysis based on VCS snapshots. Note that in our previous study [27] we looked at how much automated refactorings are interspersed with other code changes in the same commit in the aggregated data only, while in this study, we look at both automated and manual refactorings, we distinguish ten refactoring kinds, we distinguish different groups of participants, and we are able to count individual refactorings that are *completely* missing in VCS (rather than just being partially overlapped with some other changes).

We consider that a refactoring does not reach VCS if none of the AST node operations that are part of this refactoring reach VCS. An AST node operation does not reach VCS if there is another, later operation that affects the same node, up to the moment the file containing this node is committed to VCS. These non-reaching AST node operations and refactorings are essentially *shadowed* by other changes. For example, if a program entity is renamed twice before the code is committed to VCS, the first Rename refactoring is completely shadowed by the second one.

Fig. 9 shows the ratio of reaching and shadowed refactorings for the aggregated data. Since even a reaching refactoring might be partially shadowed, we also

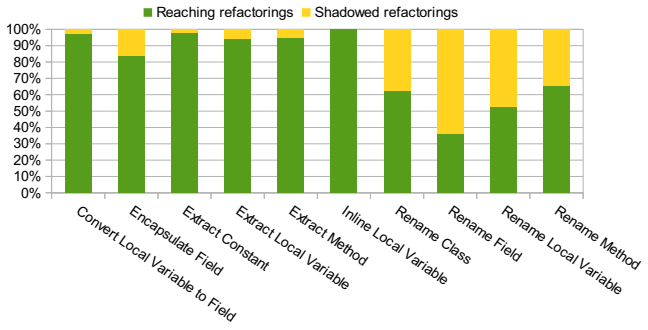


Fig. 9. Ratio of reaching and shadowed refactorings

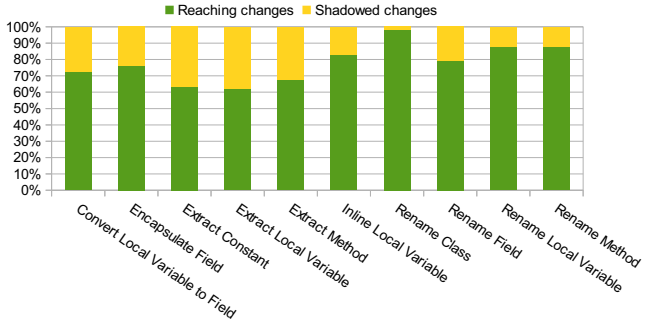


Fig. 10. Ratio of reaching and shadowed AST node operations that are part of reaching refactorings

compute the ratio of reaching and shadowed AST node operations that are part of reaching refactorings, which is shown in Fig. 10.

Our aggregated results show that for all refactoring kinds except Inline Local Variable, there is some fraction of refactorings that are shadowed. Overall, 30% of refactorings are *completely* shadowed. The highest shadowing ratio is for the Rename refactorings. In particular, 64% of the Rename Field refactorings do not reach VCS. Thus, using VCS snapshots to analyze these refactoring kinds might significantly skew the analysis results.

Although we did not expect to see any noticeable difference between manual and automated refactorings, our results show that there are significantly more shadowed manual than automated refactorings for each refactoring kind (except Inline Local Variable, which does not have any shadowed refactorings at all). Overall, 40% of manual and only 16% of automated refactorings are shadowed. This interesting fact requires further research to understand why developers underuse automated refactorings more in code editing scenarios whose changes are unlikely to reach VCS.

Another observation is that even refactorings that reach VCS might be hard to infer from VCS snapshots, since a noticeable fraction of AST node operations that are part of them do not reach VCS. This is particularly characteristic to the Extract refactorings, which have the highest ratio of shadowed AST node operations.

Our results for all groups are consistent with the aggregated results with a few exceptions. In particular, the percentage of completely shadowed refactorings for those participants who used our tool for less than 50 hours, is relatively small — 12%, which could be attributed to the fact that such participants did not have many opportunities to commit their code during the timespan of our study. Another observation is that for novice developers, the fraction of completely shadowed automated refactorings is significantly higher than the fraction of completely shadowed manual refactorings (38% vs. 10%). One of the reasons could be that novices experiment more with automated refactoring tools while learning them (e.g., they might perform an inappropriate automated refactoring and then undo it). Also, novice developers might be less confident in their refactoring capabilities and thus, try to see the outcome of an automated refactoring before deciding how (and whether) to refactor their code, which confirms our previous finding [31]. On the contrary, for the developers with more than ten years of programming experience, the amount of completely shadowed automated refactorings is very low — 2%, while the amount of completely shadowed manual refactorings is much higher — 39%. Thus, the most experienced developers tend to perform automated refactorings in code editing scenarios whose changes are likely to reach VCS.

4 Refactoring Inference Algorithm

4.1 Algorithm Overview

Inferring Migrated AST Nodes. Many kinds of refactorings that we would like to infer rearrange elements in the refactored program. To correctly infer such

refactorings, we need to track how AST nodes migrate in the program's AST. Fig. 11 shows an example of the Extract Local Variable refactoring that results in *many-to-one* migration of the extracted AST node. Fig. 12 shows the effect of this refactoring on the underlying AST. Note that the extracted AST node, string literal `"-"`, is deleted from two places in the old AST and inserted in a single place in the new AST — as the initialization of the newly created local variable.

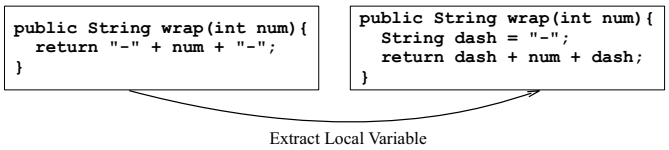


Fig. 11. An example of the Extract Local Variable refactoring that results in many-to-one migration of the extracted AST node

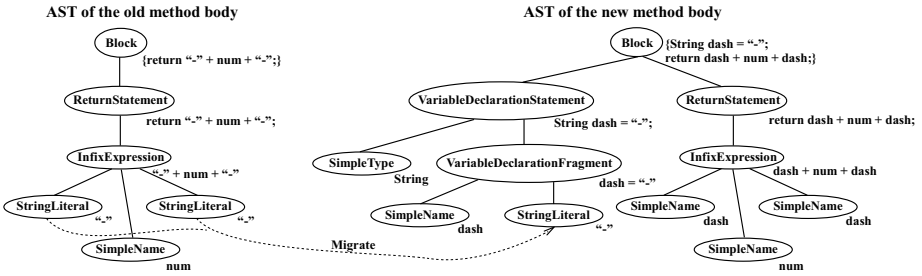


Fig. 12. The effect of the Extract Local Variable refactoring presented in Fig. 11 on the underlying AST

Our refactoring inference algorithm infers *migrate* operations from a sequence of *basic* AST node operations: *add*, *delete*, and *update*. The algorithm assigns a unique ID to each inferred *migrate* operation and marks all basic AST node operations that make part of the inferred operation with its ID.

Inferring refactorings. Our algorithm infers ten kinds of refactorings shown in Table 1. To infer a particular kind of refactoring, our algorithm looks for *properties* that are *characteristic* to it. A refactoring property is a high-level semantic code change, e.g., addition or deletion of a variable declaration. Fig. 13 shows an example of the Inline Local Variable refactoring and its characteristic properties.

Our algorithm identifies refactoring properties directly from the basic AST node operations that represent the actions of a developer. A refactoring property is described with its *attributes*, whose values are derived from the corresponding AST node operation. Our algorithm identifies 15 attributes, e.g., `entityName`,

`entityNameNodeID`, `parentID`, `migrateID`, `migratedNode`, `enclosingClassNodeID`, etc. A property may contain one or more such attributes, e.g., Migrated To Usage property has attributes `migratedNode`, `migrateID`, and `parentID`, and Deleted Entity Reference property has attributes `entityName`, `entityNameNodeID`, and `parentID`. When the algorithm checks whether a property can be part of a particular refactoring, the property's attributes are matched against attributes of all other properties that already make part of this refactoring. As a basic rule, two attributes match if either they have different names or they have the same value.

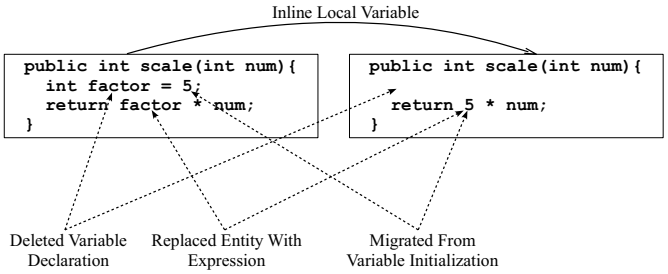


Fig. 13. An example of the Inline Local Variable refactoring and its characteristic properties

Our algorithm combines two or more closely related refactoring properties in a single refactoring *fragment*. Such fragments allow to express high level properties that could not be derived from a single AST node operation. For example, replacing a reference to an entity with an expression involves two AST node operations: *delete* entity reference and *add* expression. Consequently, the corresponding refactoring fragment, Replaced Entity With Expression, contains two properties: Migrated To Usage and Deleted Entity Reference.

The algorithm considers that a refactoring is *complete* if all its *required* characteristic properties are identified within a specific time window, which in our study is five minutes. Some characteristic properties are optional, e.g., replacing field references with getters and setters in the Encapsulate Field refactoring is optional. Also, a refactoring might include several instances of the same characteristic property. For example, an Inline Local Variable refactoring applied to a variable that is used in multiple places includes several properties of migration of the variable's initialization expression to the former usage of the variable.

Putting It All Together. Fig. 14 shows a high level overview of our refactoring inference algorithm. The algorithm takes as input the sequence of basic AST node operations marked with migrate IDs, *astNodeOperations*. The output of the algorithm is a sequence of the inferred refactorings, *inferredRefactorings*.

The refactoring inference algorithm processes each basic AST node operation from *astNodeOperations* (lines 4 – 45). First, the algorithm removes old pending complete refactorings from *pendingCompleteRefactorings* and adds them to *inferredRefactorings* (line 5). A complete refactoring is considered old if no more

```
input: astNodeOperations // the sequence of basic AST node operations
output: inferredRefactorings
1  inferredRefactoringKinds = getAllInferredRefactoringKinds();
2  inferredRefactorings = ∅; pendingCompleteRefactorings = ∅;
3  pendingIncompleteRefactorings = ∅; pendingRefactoringFragments = ∅;
4  foreach (astNodeOperation ∈ astNodeOperations) {
5   inferredRefactorings ∪= removeOldRefactorings(pendingCompleteRefactorings);
6   removeTimedOutRefactorings(pendingIncompleteRefactorings);
7   removeTimedOutRefactoringFragments(pendingRefactoringFragments);
8   newProperties = getProperties(astNodeOperation);
9   foreach (newProperty ∈ newProperties) {
10   foreach (pendingRefactoringFragment ∈ pendingRefactoringFragments) {
11    if (accepts(pendingRefactoringFragment, newProperty) {
12     addProperty(pendingRefactoringFragment, newProperty);
13     if (isComplete(pendingRefactoringFragment) {
14      remove(pendingRefactoringFragments, pendingRefactoringFragment);
15      newProperties ∪= pendingRefactoringFragment; break;
16     }
17    }
18   }
19   if (canBePartOfRefactoringFragment(newProperty) {
20    pendingRefactoringFragments ∪= createRefactoringFragment(newProperty);
21   }
22   foreach (pendingCompleteRefactoring ∈ pendingCompleteRefactorings) {
23    if (accepts(pendingCompleteRefactoring, newProperty) {
24     addProperty(pendingCompleteRefactoring, newProperty);
25     continue foreach_line9; // the property is consumed
26    }
27   }
28   foreach (pendingIncompleteRefactoring ∈ pendingIncompleteRefactorings) {
29    if (accepts(pendingIncompleteRefactoring, newProperty) {
30     newRefactoring = clone(pendingIncompleteRefactoring);
31     addProperty(newRefactoring, newProperty);
32     if (isComplete(newRefactoring) {
33      pendingCompleteRefactorings ∪= newRefactoring;
34      continue foreach_line9; // the property is consumed
35     } else pendingIncompleteRefactorings ∪= newRefactoring;
36    }
37   }
38   foreach (inferredRefactoringKind ∈ inferredRefactoringKinds) {
39    if (isCharacteristicOf(inferredRefactoringKind, newProperty) {
40     newRefactoring = createRefactoring(inferredRefactoringKind, newProperty);
41     pendingIncompleteRefactorings ∪= newRefactoring;
42    }
43   }
44  }
45 }
46 inferredRefactorings ∪= pendingCompleteRefactorings;
```

Fig. 14. Overview of our refactoring inference algorithm

properties were added to it within two minutes. Also, the algorithm removes timed out pending incomplete refactorings from *pendingIncompleteRefactorings* (line 6) as well as timed out pending refactoring fragments from *pendingRefactoringFragments* (line 7). An incomplete refactoring or a refactoring fragment times out if it was created more than five minutes ago.

In the following step, the algorithm generates refactoring properties specific to a particular AST node operation (line 8). The algorithm processes the generated properties one by one (lines 9 – 44). First, every new property is checked against each pending refactoring fragment (lines 10 – 18). If there is a refactoring fragment that accepts the new property and becomes complete, then this refactoring fragment itself turns into a new property to be considered by the algorithm (line 15). If the new property can be part of a new refactoring fragment, the algorithm creates the fragment and adds it to *pendingRefactoringFragments* (lines 19 – 21).

Next, the algorithm tries to add the new property to pending complete refactorings (lines 22 – 27). If the new property is added to a complete refactoring, the algorithm proceeds to the next new property (line 25). Otherwise, the algorithm checks whether this property can be added to pending incomplete refactorings (lines 28 – 37). If an incomplete refactoring accepts the property, it is added to a copy of this incomplete refactoring (lines 30 – 31). If adding the new property makes the new refactoring complete, it is added to *pendingCompleteRefactorings* (line 33) and the algorithm proceeds to the next new property (line 34). Otherwise, the new refactoring is added to *pendingIncompleteRefactorings* (line 35).

If the new property does not make any of the pending incomplete refactorings complete, the algorithm creates new refactorings of the kinds that the new property is characteristic of and adds these new refactorings to *pendingIncompleteRefactorings* (lines 38 – 43).

Finally, after processing all AST node operations, the algorithm adds to *inferredRefactorings* any of the remaining pending complete refactorings (line 46).

More details about our algorithm, including the full list of properties and their component attributes as well as composition of refactorings and refactoring fragments, can be found in our technical report [26].

4.2 Evaluation of Refactoring Inference Algorithm

Unlike the authors of the other two similar tools [9, 14], we report the accuracy of our continuous refactoring inference algorithm on *real world* data. First, we evaluated our algorithm on the automated refactorings performed by our participants, which are recorded precisely by Eclipse. We considered 2,398 automated refactorings of the nine out of the ten kinds that our algorithm infers (we disabled the inference of the automated Encapsulate Field refactoring in our experiment because the inferencer did not scale for one participant, who performed many such refactorings one after another). A challenge of any inference tool is to establish the *ground truth*, and we are the first to use such a large ground truth. Our algorithm correctly inferred 99.3% of these 2,398 refactorings. The uninferred 16 refactorings represent unlikely code editing scenarios, e.g., ten of them are

the Extract Local Variable refactorings in which Eclipse re-writes huge chunks of code in a single shot.

Also, we randomly sampled 16.5 hours of code development from our corpus of 1,520 hours. Each sample is a 30-minute chunk of development activity, which includes writing code, refactoring code, running tests, committing files, etc. To establish the ground truth, the second author manually replayed each sample and recorded any refactorings (of the ten kinds that we infer) that he observed. He then compared this to the numbers reported by our inference algorithm. The first and the second authors discussed any observed discrepancies and classified them as either false positives or false negatives. Table 5 shows the sampling results for each kind of the refactoring that our algorithm infers. Overall, our inference algorithm has a precision of 0.93 and a recall of 1.

Table 5. Sampling results

Refactoring	True positives	False negatives	False positives
Convert Local Variable to Field	1	0	1
Encapsulate Field	0	0	0
Extract Constant	0	0	0
Extract Local Variable	8	0	0
Extract Method	2	0	1
Inline Local Variable	2	0	0
Rename Class	3	0	0
Rename Field	5	0	0
Rename Local Variable	28	0	2
Rename Method	4	0	0
Total	53	0	4

5 Threats to Validity

5.1 Experimental Setup

We encountered difficulties in recruiting a larger group of experienced programmers due to issues such as privacy, confidentiality, and lack of trust in the reliability of research tools. However, we managed to recruit 23 participants, which we consider a sufficiently big group for our kind of study. Our dataset is not publicly available due the non-disclosure agreement with our participants.

Our dataset is non-homogeneous. In particular, our participants have different affiliations, programming experience, and used CodingTracker for a various amount of time. To address this limitation, we divided our participants in seven groups along these three categories. We answered each research question for every group as well as for the aggregated data and reported the observed insignificant discrepancies.

Our results are based on the code evolution data obtained from developers who use Eclipse for Java programming. Nevertheless, we expect our results to generalize to *similar* programming environments.

We infer only ten kinds of refactorings, which is a subset of the total number of refactorings that a developer can apply. To address this limitation to some extent, we inferred those refactoring kinds that are previously reported as being the most popular among automated refactorings [31].

5.2 Refactoring Inference Algorithm

Our refactoring inference algorithm takes as input the basic AST node operations that are inferred by another algorithm [27]. Thus, any inaccuracies in the AST node operations inference algorithm could lead to imprecisions in the refactoring inference algorithm. However, we compute the precision and recall for both these algorithms applied together, and thus, account for any inaccuracies in the input of the refactoring inference algorithm.

Although the recall of our refactoring inference algorithm is very high, the precision is noticeably lower. Hence, some of our numbers might be skewed, but we believe that the precision is high enough not to undermine our general observations.

To measure the precision and recall of the refactoring inference algorithm, we sampled around 1% of the total amount of data. Although this is a relatively small fraction of the analyzed data, the sampling was random and involved 33 distinct 30-minute intervals of code development activities, thus a manual analysis of 990 minutes of real code development.

6 Related Work

6.1 Empirical Studies of Refactoring Practice

The practice of refactoring plays a vital role in software evolution and is an important area of research. Studies by Xing and Stroulia [33], and Dig and Johnson [5] estimate that 70 – 80% of all code evolution can be expressed as refactorings.

Murphy et al. [23] were the first to study the usage of automated refactoring tools. Their study provided the first empirical ranking of the relative popularities of different automated refactorings, demonstrating that some tools are used more frequently than others. Subsequently, Murphy-Hill et al.'s [25] study on the use of automated refactoring tools provided valuable insights into the use of automated refactorings in the wild by analyzing data from multiple sources.

Due to the non-intrusive nature of CodingTracker, we were able to deploy our tool to more developers for longer periods of time, providing a more complete picture of refactoring in the wild. We inferred and recorded an *order* of magnitude more manual refactoring invocations compared to Murphy-Hill et al.'s sampling-based approach. Murphy-Hill sampled 80 commits from 12 developers for a total of 261 refactoring invocations whereas our tool recorded 1,520 hours from 23 developers for a total of 5,371 refactoring invocations.

Murphy-Hill et al.'s [25] study found that (i) refactoring tools are underused and (ii) the kinds of refactorings performed manually are different from those performed using tools. Our data (see **RQ3**) corroborates both these claims. Due to the large differences in the data sets (261 from Murphy-Hill et al. vs. 5,371 from ours), it is infeasible to meaningfully compare the raw numbers for each refactoring kind. Our work also builds upon their work by providing a more detailed breakdown of the manual and automated usage of each refactoring tool according to different participant's behavior.

Vakilian et al. [30] observed that many advanced users tend to compose several refactorings together to achieve different purposes. Our results about clustered refactorings (see **RQ6**) provide additional empirical evidence of such practices.

6.2 Automated Inference of Refactorings

Traditionally, automated refactoring inference relies on comparing two different versions of source code and describing the changes between versions of code using higher-level *characteristic properties*. A refactoring is detected based on how well it matches a set of characteristic properties. Our previous tool, *RefactoringCrawler* [6], used references of program entities (instantiations, method calls, and type imports) as its set of characteristic properties. Weißgerber and Diehl [32] used names, signature analysis, and clone detection as their set of characteristic properties. More recently, Prete et al. [28] devised a template-based approach that can infer up to 63 of the 72 refactorings cataloged by Fowler [10]. Their templates involve characteristic properties such as accesses, calls, inherited fields, etc., that model code elements in Java. Their tool, *Ref-Finder*, infers the widest variety of refactorings to date.

All these approaches rely exclusively on VCS snapshots to infer refactorings. We have shown in **RQ7** that many refactorings do not reach VCS. This compromises the accuracy of inference algorithms that rely on snapshots. To address such inadequacies, our inference algorithm leverages fine-grained edits. Similar to existing approaches, our algorithm infers refactorings by *matching* a set of characteristic properties for each refactoring. In contrast to existing approaches, our properties are precise because they are constructed directly from the AST operations that are recorded on each code edit.

In parallel with our tool, Ge et al. [14] developed *BeneFactor* and Foster et al. [9] developed *WitchDoctor*. Both these tools continuously monitor code changes to detect and complete manual refactorings in *real-time*. Though conceptually similar, our tools have different goals — we infer complete refactorings, while *BeneFactor* and *WitchDoctor* focus on inferring and completing partial refactorings in real time. Thus, their tools can afford to infer fewer kinds of refactorings and with much lower accuracy. Nonetheless, both highlight the potential of using refactoring inference algorithms based on fine-grained code changes to improve the IDE. We compare our tool with the most similar tool, *WitchDoctor*, in more detail in our technical report [26].

7 Conclusions

There are many ways to learn about the practice of refactoring, such as observing and reflecting on one's own practice, observing and interviewing other practitioners, and controlled experiments. But an important way is to analyze the changes made to a program, since programmers' beliefs about what they do can be contradicted by the evidence. Thus, it is important to be able to analyze programs and determine the kind of changes that have been made. This is traditionally done by looking at the difference between snapshots. In this paper, we have shown that VCS snapshots lose information. A continuous analysis of change lets us see that refactorings tend to be clustered, that programmers often change the name of an item several times within a short period of time and perform more manual than automated refactorings.

Our algorithm for inferring change continuously can be used for purposes other than understanding refactoring. We plan to use it as the base of a programming environment that treats changes intelligently. Continuous analysis is better at detecting refactorings than analysis of snapshots, and it ought to become the standard for detecting refactorings.

Acknowledgments. We would like to thank Nathan Hirtz for guiding us in statistical sampling techniques. We also thank Darko Marinov and students in the Software Engineering seminar at UIUC for insightful comments on earlier drafts of this paper. This work was partially supported by the National Science Foundation grants number CCF-1117960 and CCF-1213091.

References

1. Antoniol, G., Penta, M.D., Merlo, E.: An automatic approach to identify class evolution discontinuities. In: IWPSE (2004)
2. Bansiya, J.: Evaluating application framework architecture structural and functional stability. In: Object-Oriented Application Frameworks: Problems and Perspectives (1999)
3. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Professional (2004)
4. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: OOPSLA (2000)
5. Dig, D., Johnson, R.: The role of refactorings in API evolution. In: ICSM (2005)
6. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 404–428. Springer, Heidelberg (2006)
7. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. Journal of Soft. Maint. and Evol.: Research and Practice (2006)
8. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE TSE (2001)
9. Foster, S., Griswold, W.G., Lerner, S.: WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In: ICSE (2012)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc. (1999)

11. Gall, H., Hajek, K., Jazayeri, M.: Detection of logical coupling based on product release history. In: ICSM (1998)
12. Gall, H., Jazayeri, M., Klsch, R.R., Trausmuth, G.: Software evolution observations based on product release history. In: ICSM (1997)
13. Gall, H., Jazayeri, M., Krajewski, J.: CVS release history data for detecting logical couplings. In: IWMPSE (2003)
14. Ge, X., DuBose, Q.L., Murphy-Hill, E.: Reconciling manual and automatic refactoring. In: ICSE (2012)
15. Godfrey, M.W., Zou, L.: Using origin analysis to detect merging and splitting of source code entities. IEEE TSE (2005)
16. Henkel, J., Diwan, A.: CatchUp!: Capturing and replaying refactorings to support API evolution. In: ICSE (2005)
17. Kim, M., Cai, D., Kim, S.: An empirical investigation into the role of API-level refactorings during software evolution. In: ICSE (2011)
18. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: FSE (2012)
19. Kim, S., Pan, K., Whitehead Jr., E.J.: When functions change their names: Automatic detection of origin relationships. In: WCRE (2005)
20. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc. (1985)
21. Mattson, M., Bosch, J.: Frameworks as components: a classification of framework evolution. In: NWPER (1998)
22. Mattson, M., Bosch, J.: Three Evaluation Methods for Object-oriented Frameworks Evolution - Application, Assessment and Comparison. Tech. rep., University of Karlskrona/Ronneby, Sweden (1999)
23. Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE? IEEE Software (2006)
24. Murphy-Hill, E., Black, A.P.: Breaking the barriers to successful refactoring: observations and tools for extract method. In: ICSE (2008)
25. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE TSE (2012)
26. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: Using Continuous Code Change Analysis to Understand the Practice of Refactoring. Tech. Rep. (2012), http://hdl.handle.net/2142/33783
27. Negara, S., Vakilian, M., Chen, N., Johnson, R.E., Dig, D.: Is it dangerous to use version control histories to study source code evolution? In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 79–103. Springer, Heidelberg (2012)
28. Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: ICSM (2010)
29. Rysselberghe, F.V., Demeyer, S.: Reconstruction of successful software evolution using clone detection. In: IWPSE (2003)
30. Vakilian, M., Chen, N., Moghaddam, R.Z., Negara, S., Johnson, R.E.: A compositional paradigm of automating refactorings. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 527–551. Springer, Heidelberg (2013)
31. 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)
32. Weißgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE (2006)
33. Xing, E.Z., Stroulia: Refactoring practice: How it is and how it should be supported - an eclipse case study. In: ICSM (2006)
34. Xing, Z., Stroulia, E.: Analyzing the evolutionary history of the logical design of object-oriented software. TSE (2005)

What Programmers Do with Inheritance in Java

Ewan Tempero[1], Hong Yul Yang[1], and James Noble[2]

[1] University of Auckland, Auckland, New Zealand
e.tempero,hongyul@cs.auckland.ac.nz
[2] School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand
kjx@ecs.vuw.ac.nz

Abstract. Inheritance is a distinguishing feature of object-oriented programming languages, but its application in practice remains poorly understood. Programmers employ inheritance for a number of different purposes: to provide subtyping, to reuse code, to allow subclasses to customise superclasses' behaviour, or just to categorise objects. We present an empirical study of 93 open-source Java software systems consisting over over 200,000 classes and interfaces, supplemented by longitudinal analyses of 43 versions of two systems. Our analysis finds inheritance is used for two main reasons: to support subtyping and to permit what we call external code reuse. This is the first empirical study to indicate what programmers do with inheritance.

1 Introduction

Inheritance is a concept that is given significant visibility to those learning object-oriented design, with texts on object-oriented programming often devoting several chapters to the subject (e.g. [4]). This raises the question of to what degree is inheritance actually used. In previous work we measured how much inheritance was used in a software system in terms of *how often* a developer *made the decision* to create an inheritance relationship between two types [28]. What we found was, on average, 3 out of 4 types were defined using some form of inheritance: inheritance is clearly important within Java programs.

Our results however do not tell the full story with regards inheritance use. While they tell us how much inheritance is used, they do not tell us what the designer uses it for, they do not tell us to what degree its use is *necessary*. It may be that some of the use we have observed is not appropriate use of inheritance. The main goal of the study presented in this paper is to determine whether or not this is the case. If the use is mainly appropriate, then this is important to know for two reasons. The first is that our earlier results become much more relevant in demonstrating the importance of inheritance. The second reason is the systems we analysed provide a benchmark for how inheritance is used.

Our previous study simply measured the amount of inheritance in programs, that is, what inheritance relationships exist between types. We only had to look at the `extends` and `implements` clauses of type declarations. In this study, we must look

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 577–601, 2013.

at the implementation of each method to understand how those inheritance relationships were actually exercised. Whereas our last study considered the question "How do programs use inheritance?", in this study our question is **"What do programmers do with inheritance?"**, that is, having made the decision to use inheritance at the design level, what benefits follow from the use of inheritance. We are particularly interested in discovering unnecessary uses of inheritance, that is, cases where an inheritance relationship exists, but where it is not required. We address this question by studying a large corpus of open source Java systems.

This paper makes the following contributions:

- We develop a model of inheritance that represents how inheritance is used.
- We present the results of evaluating a corpus of open source Java systems against our model of inheritance, and make our data set available.
- Our overall conclusion is that inheritance in Java is mostly used to support subtyping and to permit external reuse. Additionally, a significant fraction of subclasses rely on polymorphic self-calls to customise their superclasses' methods' behaviour.

The rest of the paper is organised as follows. In the next section, we identify four research questions by considering previous discussions of why inheritance is used. Section 3 discuses inheritance in Java in detail, a necessary precursor to section 4, which presents our model of inheritance and the methodology of our study. In section 5 we present our results, and discuss their consequences in section 6. Finally, we present conclusions and discuss future work in section 7.

2 Motivation and Research Questions

There is much discussion in the research literature and trade press on inheritance as it applies to software, but there seems to be little on understanding how it is actually used. There seems to be a considerable uncertainty as to what it is or how to use it, if the number of web sites, blogs, and articles in the trade press are anything to go by. At the same time, there are quite public criticisms of inheritance, through writings with provocative titles such as "Why extends is evil" [15] or "Inheritance is evil, and must be destroyed" [24]. Examining such criticisms, we might conclude they are overstating the case based on a small set of examples, as there is little objective evidence that the problems they identify are widespread. Nevertheless, authors such as Gamma et al. instruct us to "Favor object composition over class inheritance" [12], which suggests they at least have seen enough questionable use of inheritance as to prompt such advice.

Within the research community, the recent focus has been on measuring inheritance with the hope of understanding the relationship between its use and some notion of quality of the software. For example Chidamber and Kemerer introduced the DIT and NOC metrics that measure two aspects of a individual class' use of inheritance [8,7]. There have been several studies to establish the relationship between measurements from these and similar metrics and quality attributes such as maintenance ([18]) or prediction of fault prone classes [1,3].

We detail three further related studies, those by Daly et al. [10], Cartwright [6], and Harrison et al [14]. Daly et al. examined the impact of depth of inheritance (using DIT)

on maintenance, with the conclusion that inheritance had a negative effect on maintenance time. In a rare replication Cartwright carried out a similar study, with results suggesting that depth of inheritance had a positive effect on maintenance. Another replication was carried out by Harrison et al. Their results suggest that depth of inheritance made it harder to modify systems, but that size and functionality of a system may affect understandability more than the "amount of inheritance" used.

There could be several explanations for these inconsistent results. For example, it could be that the systems under study were too small for inheritance to be the main factor affecting maintenance effort. It could also be possible that the uses of inheritance were not the same in all studies, or that depth of inheritance is not sufficient to characterise how inheritance is used. For example, as we have previously reported, different uses of overriding could explain the variation [27]. Another possibility, which we explore in this paper, is that programmers choose to use inheritance for different reasons. It could have been that the systems in the different studies used inheritance for different purposes, meaning the studies were not in fact comparing like with like.

In the early discussions of inheritance, there was much debate as to how it could be used, how languages should provide it, and whether it was even a good idea. Inheritance has been the subject of much discussion within the research community, The discussion explored such things as its interaction with encapsulation [23], how type systems are affected by inheritance [9], how it relates to other features such as genericity [19], or whether variants such as multiple inheritance are worth having [5,30].

Of particular interest to us are two reports of how inheritance is or can be used. Meyer described what he regarded as 12 different valid uses of inheritance [20]. Taivalsaari discussed the many varieties and uses of inheritance, and provided a taxonomy for analysing inheritance [25]. Taivalsaari also observed that there seemed to be many benefits compared to other programming language features, but – crucially — described inheritance "*an incremental modification mechanism in the presence of late-bound self-reference*" and concluded that this seemed to be its most profound benefit. This conclusion is interesting. Late-bound self-reference — that method invocations on `this` (in Java) are also polymorphic as with any method invocation — is a feature of object-oriented languages that usually does not get much attention, and in some cases not much language support (e.g. Go [22]). This gives us our first research question:

RQ1: To what extent is late-bound self-reference relied on in the designs of Java systems?

Neither Taivalsaari nor Meyer provided empirical evidence to support their conclusions, and some of the uses of inheritance they described have no obvious operationalisation. These papers are also now quite old, and so it could be argued may not be relevant now, however they provide a useful starting point for understanding inheritance. In particular, Taivalsaari's taxonomy identifies three dimensions for analysing inheritance — what he called incremental modification, property inheritance, and interface inheritance. This provides a good basis for an empirical study, as we discuss in the next section.

Taivalsaari also observes that one view of inheritance is that it supports conceptual specialisation, but he and others have observed that most languages allow a class that inherits to almost arbitrarily change its behaviour, and so in such cases the inheritance

relationship would not reflect true conceptual specialisation. He comments, however, that "Subtyping, on the other hand, expresses conceptual specialization" and summarises the then thinking on the relationship between inheritance and subtyping. In Java, the subtype relationship is expressed using the Java inheritance mechanisms, and we will discuss this in more detail in the next section.

Taivalsaari also comments "In fact, the use of inheritance for conceptual specialization seems to be an ideal that is rarely realized" referring to Smalltalk and C++ libraries of the day. This is a surprising claim, as if true it would mean that the subtype relationship is "rarely" used. This has not been our personal experience with Java code, and in fact there is advice advocating using inheritance for subtyping. For example, Bloch says "Inheritance is appropriate only in circumstances where the subclass really is a *subtype* of the superclass" [2](p85). We know of no empirical evidence to support or refute such a claim. This leads to our next question:

RQ2: To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to the design?

We will discuss the details of what it means to be "necessary" in section 4.

While Meyer (and to a lesser extent Taivalsaari) discuss a number of ways inheritance might be used, contemporary advice seems to be more conservative. As noted above, Gamma et al. caution against some forms of use. Bloch repeats the advice "Favor composition over inheritance" [2](Item 16) and provides a compelling example to support this. The advice is based on the argument that the form of inheritance referred to by Gamma et al. and Bloch fundamentally is an implementation decision. As such, inheritance breaks encapsulation, as observed previously by Snyder [23]. In fact Bloch shows a mechanical procedure to convert from this use of inheritance to composition (replacing inheritance with delegation).

Given that advice by such prominent authors is to avoid inheritance where possible, we might expect that there is infrequent use of the form of inheritance they refer to. Specifically, we might expect that inheritance is avoided in favour of composition. It is difficult to tell when something is being avoided, but we can tell when it is not, so for our next research question we ask:

RQ3: To what extent can inheritance be replaced by composition?

There has been other discussion regarding use of inheritance either directly or indirectly. For example, Johnson and Foote discuss how features of object-oriented languages, including inheritance, can be used to develop reusable code [16]. In a similar vein, various specific uses have been recorded [12,13]. We do not repeat this work, but are interested in identifying any inheritance idioms in common use:

RQ4: What other inheritance idioms are in common use in Java systems?

3 Understanding Inheritance

In order to measure how inheritance is used, we need to understand what it means. The study we present is of Java code, and so some of the details are Java specific.

For example, by "inheritance" we mean when a Java type (class, interface, annotation, enum) `extends` or `implements` another type. While the details are Java specific, we believe the general concepts apply to most object-oriented languages.

```
class P {                          class C extends P {
  void p() {                         void c() {
    q();                                 q();// internal
  }                                    }
  void q() {                         }
    ...
  }
}
class M {                          class D extends P {
  void m(P aP) {                     void q() {
    aP.p();                              ...
  }                                    }
}                                  }

class N {
  void useReuse() {
    C aC = new C();
    aC.p();                         // external
  }
  void useSubtype() {
    M anM = new M();
    C aC = new C();
    anM.m(aC);                     // subtype
    D aD = new D();
    aD.p();                        // downcall when executing D#p()
  }
}
```

Fig. 1. Uses of Inheritance: "external reuse", "internal reuse", and "subtype". Modifiers have been elided.

Inheritance is often presented as what Taivalsaari refers to as "property inheritance" — one class (the *child*) acquires properties of another (the *parent*) by inheriting them. This is one way in which inheritance supports **reuse**; the inheriting class can be written faster because the inherited code does not have to be rewritten. This is illustrated in Figure 1. In the method `N#useReuse()` the method `p()` is invoked on an instance of `C`, however the code that is actually executed was not written for `C` but `C` has acquired it through inheriting (extending) `P`. Note that `p()` was accessed from outside the class `C`, which we refer to as **external reuse**. The method `C#c()` also makes use of an inherited method, but does so from within `C`, which we refer to as **internal reuse**. Figure 2 shows a more well-known example of internal reuse.

For external or internal reuse, every access to a member of a type is examined. If the member is not declared in that type, then it is some form of reuse. If the type that the

```
class Stack<E> extends Vector<E> {
  ...
  public E push(E item) {
    addElement(item);
    return item;
  }
  ...
}
```

Fig. 2. `Stack` demonstrates internal reuse by making a "self-call" (in bold) on its parent `Vector` as part of the implementation of one of its methods

member is declared in is an ancestor of type containing the code where the access takes place, then it is internal reuse, otherwise it is external reuse.

If a child class has all of the properties of a parent class, then it seems reasonable to expect that an object from the child can be used wherever an object from the parent is expected. This is Taivalsaari's interface inheritance dimension, although it is perhaps best known as the Liskov Substitution Principle [17]. This ability to substitute child objects for parent objects is formally recognised in the Java type system by regarding the type associated with the child class to be a **subtype** of that associated with the parent class. In the method `N#useSubtype()` in Figure 1, an instance of `C` is legally passed to the method `M#m(P)`, even though that method expects an instance of `P`. Without the subtype relationship, the code in `N#useSubtype()` would have to be duplicated in order to handle types other than `P`.

As noted in the previous section, many languages, Java included, allow the inheriting class to change what it inherits. The mechanism for doing so (for methods) is *overriding*. While doing so can result in inheritance no longer corresponding to conceptual specialisation, it can also allow quite sophisticated behaviour to be described. Taivalsaari includes this in his incremental modification dimension, but it is the late-bound self-reference aspect of it that is of interest to us. In Figure 1, when `aD.p()` in `N#useSubtype()` executes, it invokes `P#p()` which in turn uses the (implied) self-reference to invoke `q()`. However in this case, the late binding of the self-reference means that it is actually `D#q()` that is called. Late bound self-reference means one method can call another method "below" it in the inheritance hierarchy, which we refer to as a **downcall** for brevity. For a downcall to take place, a method in a class must make a self-call to another method, that other method must be overridden by a descendant class, and the calling method must be invoked on an object of the descendant class.

The uses described above come from the standard descriptions of inheritance. We are aware of other possibilities. One idiom is the **constants** interface, where an interface is a repository of useful constants, and any class "implementing" it can use those constants without the need to qualify them. While this practice is no longer recommended [2], it does represent a use of inheritance. Another idiom is the so-called "marker" interface. Such interfaces have no members, but it is necessary that classes implement to indicate they have certain capabilities that have no associated methods.

4 Methodology

Our goal is to determine why developers have made the decision to create an inheritance relationship between two types, that is, what *purpose* did they likely have in mind? We can infer this by examining the use they make of the relationship. We then measure the uses with respect to a collection of systems.

4.1 Modelling Inheritance

Metrics can be defined by different means [11]. One means is to base the definitions on a model of what is to be measured. The measurements we present in this work come from metrics based on the model of how inheritance is used in software, which we call the *inheritance graph*.

Measurement assigns numbers to attributes of entities, and it is the entities that we model. We want to measure the software that makes up what we generically refer to as a "system," however defining exactly what this is difficult, as we have discussed elsewhere [26]. To resolve these difficulties we consider a system to be just those types that were created for that system. This excludes the Java Standard API and third-party libraries, a decision whose consequences we discuss further below.

For this study, we limit the types to just classes and interfaces, and furthermore, for classes we do not include exceptions (generally, any type that is a descendant of `java.lang.Throwable`). Enums, annotations, and exceptions are all defined using inheritance and so this use of inheritance is not a choice by the developer.

For a given system, its inheritance graph is a directed graph where the vertices represent any type (class or interface) associated with the system implementation and the edges connect any pair of types that have some kind of explicit inheritance relationship (`extends` or `implements`), that is, the relationship is described in the code. This means we do not model edges between system types and non system types (third-party code). The vertices are named by the fully qualified name of the type they represent.

The edges have a set of attributes that capture the information for this study, which is defined below. Direct metrics are then defined in terms of boolean expressions describing the presence or absence of attributes on edges. Some attributes represent properties inherent in the system code, some represent what we have observed in the system code in terms of why the inheritance relationships are needed, and some represent information that has been established only by heuristics. We will indicate which applies when necessary. In the interests of brevity, we will uses phrases such as "subtype edges", by which we mean "inheritance relationships that we observed were relied on for the purpose of supporting the subtype relationship."

CC, CI, II: An edge will have one of these attributes if it represents, respectively, a Class-Class (`extends`), a Class-Interface (`implements`), or an Interface-Interface (`extends`) relationship between system types.

External Reuse: An edge from type `S` (child) to `T` (parent) has the external reuse attribute if there is a class `E` that has no inheritance relationship with `T` (or `S`), it invokes a method `m()` or accesses a field `f` on an object declared to be of type `S`, and `m()` or `f` is a member (possibly inherited) of `T`.

The class E is using a member of S that was not declared in S, which is only possible because S has an inheritance relationship with T, so the inheritance relationship is necessary for this to be possible. This definition does not assume S and T are classes, but we only discuss external reuse with respect to classes in this paper.

Internal Reuse: An edge from classes A (child) to B (parent) has the internal reuse attribute when a method declared in A invokes a method m() or accesses a field f on an object constructed from A and m() or f is a member (possibly inherited)of B.

Without the stated inheritance relationship, it would not be possible to invoke m() or access f in this way.

Subtype: An edge from types S (child) to T (parent) has the subtype attribute when there is a class E (which could be S or T) in which an object of type S is supplied where an object of type T, or a supertype of T, is expected.

Within E, this might be assigning an object of type S to a variable declared to be type T, passing an actual parameter of type S to a formal parameter of type T, returning an object of type S when the formal return type is T, or casting an expression of type S to type T.

Without the stated inheritance relationship, S would not be a subtype of T, and so the substitution would not be possible. This means that this relationship is necessary in order for the code to compile.

Downcall: An edge from classes C (child) to D (parent) has the downcall attribute when a method d() that is a member (possibly inherited) of D invokes a method m() as a self-call that is declared in C.

The inheritance relationship is necessary for d() to invoke m(). The method m() must be declared in D or an ancestor of D, so d() is making a self-call to m(), but C overrides that declaration. The object on which the invocation takes place must be constructed from C or one of its descendants.

Framework: An edge from types P to Q that does *not* have external reuse, internal reuse, subtype, or downcall, has the framework attribute if Q is a descendant of a third-party type. (See also Section 4.3.)

Constants: An edge from types E to F has the constants attribute if F has only fields declared in it and the fields are constants (`static final`), and all outgoing edges (if any) from F either have the constants attribute or are to `java.lang.Object`. The type F can be either an interface or a class.

Marker: An edge from type G to interface H has the marker attribute if H has nothing declared in it and all outgoing edges (if any) from H have the marker attribute.

Super: An edge from class K to class L has the super attribute if a constructor for K explicitly invokes a constructor in L via `super`. (See also Section 4.2.)

Generic: An edge from type R to type S has the generic attribute if there is a cast from `Object` to S and there is an edge from R to some (non-`Object`) type T (See also Section 4.3).

4.2 Study Details

We studied the 93 open-source Java systems from the 20101126 release of the Qualitas Corpus [26] listed in Figure 3. Not all systems from this release of the corpus were

ant-1.8.1 antlr-3.2 aoi-2.8.1 argouml-0.30.2 aspectj-1.6.9 axion-1.0-M2 c_jdbc-2.0.2 castor-1.3.1 cayenne-3.0.1 checkstyle-5.1 cobertura-1.9.4.1 colt-1.2.0 columba-1.0 derby-10.6.1.0 displaytag-1.2 drawswf-1.2.9 drjava-stable-20100913-r5387 emma-2.0.5312 exoportal-v1.0.2 findbugs-1.3.9 fitjava-1.1 fitlibraryforfitnesse-20100806 freecol-0.9.4 freecs-1.3.20100406 galleon-2.3.0 ganttproject-2.0.9 heritrix-1.14.4 hibernate-3.6.0-beta4 hsqldb-2.0.0 htmlunit-2.8 informa-0.7.0-alpha2 ireport-3.7.5 itext-5.0.3 jFin_DateMath-R1.0.1 james-2.2.0 jasml-0.10 javacc-5.0 jchempaint-3.0.1 jedit-4.3.2 jext-5.0 jfreechart-1.0.13 jgraph-5.13.0.0 jgraphpad-5.10.0.2 jgrapht-0.8.1 jgroups-2.10.0 jhotdraw-7.5.1 jmeter-2.4 jmoney-0.4.4 joggplayer-1.1.4s jparse-0.96 jpf-1.0.2 jrat-0.6 jre-1.5.0_22 jrefactory-2.9.19 jruby-1.5.2 jsXe-04_beta jspwiki-2.8.4 jtopen-7.1 jung-2.0.1 junit-4.8.2 log4j-1.2.16 lucene-2.4.1 marauroa-3.8.1 maven-3.0 megamek-0.35.18 mvnforum-1.2.2-ga myfaces_core-2.0.2 nakedobjects-4.0.0 nekohtml-1.9.14 openjms-0.7.7-beta-1 oscache-2.4.1 picocontainer-2.10.2 pmd-4.2.5 poi-3.6 pooka-3.0-080505 proguard-4.5.1 quickserver-1.4.7 quilt-0.6-a-5 roller-4.0.1 rssowl-2.0.5 sablecc-3.1 springframework-1.2.7 squirrel_sql-3.1.2 struts-2.2.1 sunflow-0.07.2 tapestry-5.1.0.5 tomcat-7.0.2 trove-2.1.0 velocity-1.6.4 webmail-0.7.10 weka-3.7.2 xalan-2.7.1 xerces-2.10.0

Fig. 3. Systems studied, including version identifier

included as the tools we used had memory limitations that restricted the size of the systems that we could analyse. Table 1 gives provides statistics of some of these systems.

We also studied the history of two systems: `ant`, with 20 releases from version 1.1 to 1.8.1, and `freecol`, with 23 releases from version 0.3.0 to 0.9.4. We chose these two systems due to having the data for all releases in the corpus, and because they come from quite different domains (`ant` is a build tool with a plug-in architecture and managed through XML documents; `freecol` is a strategy game with a graphical interface, multi-media components, client-server architecture, and network communication).

The details of the systems can be found on the corpus website[26], in particular details of how we identified types belonging a given system. We analysed the bytecode of these systems. While most of what was needed for the analysis is in the bytecode, there is some loss of information as discussed below (Section 4.3).

There are several different kinds of analysis performed. To determine the subtype attribute, we first examine the code to find where substitution can occur. The specific cases we detect are: passing a parameter, returning a value, assignment, and cast. For example, if the declared return type of a method is `T`, but the `return` statement references a variable of a different type `S`, then there must be a subtype relationship between `S` and `T`, a relationship that is necessary for the code to compile. An example of the parameter passing case is shown in method `N#useSubtypes` of Figure 1, where an object of type `C` is passed to a method whose formal parameter type is `P`.

The subtype analysis uses what is essentially a reachability analysis for all reference type definitions to all uses. Where the type of the def does not match the type of the use, and since we know the code compiled (as we are analysing bytecode), we know we are dealing with subtype use. The tool we use is based on the Soot framework [29].

Having identified when subtype substitution is used, we then match the relationships required to the relationships expressed in the code. This is necessary because those

Table 1. Statistics for representative subset of systems studied (version elided). **Types** — number of types (including nested) in the system; **KLOC** — non-commented non-blank lines of code (thousands); **CC**, **CI**, **II** — number of the respective kinds of edges.

System	Types	KLOC	CC	CI	II
ant	1202	108	672	290	18
aspectj	3127	412	1142	626	110
derby	2755	593	697	525	91
drjava	5051	62	1269	1119	86
fitjava	85	2	43	0	0
freecol	1542	82	392	127	3
jrat	255	14	34	37	0
jre	11736	831	5735	5102	799
jruby	5783	160	3671	735	12
jsXe	144	9	11	3	0
jtopen	3482	397	1347	687	16
megamek	2969	259	1283	213	10
mvnforum	4194	51	18	45	0
nakedobjects	4963	110	1307	732	349
nekohtml	2016	7	10	8	0
trove	715	2	126	269	0
weka	2125	224	516	1047	10

```
class MapboardAction extends FreeColAction {
  ...
}
class LoadAction extends MapboardAction {
  ...
}
```

Fig. 4. Classes from `freecol` illustrating the need to use transitivity to determine the subtype attribute. (Package name and modifiers elided)

required may not be explicit. For example, given the declarations from `freecol` shown in Figure 4, it is possible to substitute `LoadAction` for `FreeColAction` despite the fact that there is no direct subtype relationship between these two types. Any code that depends on such a substitution being possible will result in the edges representing the `LoadAction – MapboardAction` and `MapboardAction – FreeColAction` relationships being given the subtype attribute.

There are also two special cases that must be addressed. One is the "sideways" cast, where Java allows what looks like a cast between unrelated types. In the example in Figure 5, the cast will be successful provided an instance of `C` is passed to the method. Such situations represent use of the subtype relationship between `C` and its parents, and so must be detected in order to correctly identify all subtype uses.

The other case involves the pseudo variable `this`, which can change its type in the presence of inheritance. This change can indicate the use of a subtype relationship. In

```
interface SidewaysA {}
interface SidewaysB {}
class SidewaysC implements SidewaysA, SidewaysB {}
class SideWays {
    public void demo(SidewaysA sa) {
        SidewaysB sb = (SidewaysB) sa;
    }
}
```

Fig. 5. Example of a "sideways" cast

```
class P {
  // A constructor expects type P
  private A anA = new A(this);
}
class C extends P {
  // C is passed to A constructor
}
```

Fig. 6. Example showing `this` changing type

the example in Figure 6, the use of `this` in the constructor call to `A` indicates the use of the subtype relationship between `P` and `C`.

For external or internal reuse, every access to a member of a type is examined. If the member is not declared in that type, then it is some form of reuse. If the type that the member is declared in is an ancestor of the type containing the code where the access takes place, then it is internal reuse, otherwise it is external reuse.

If a method invocation is a self-call (whether on a method declared in that class or due to internal reuse), then it could be a downcall. In such cases, all descendants are examined and if the method being invoked is overridden, then we make the conservative assumption that a downcall can take place.

When doing the analysis, we ignore the use of default constructors. There must always be a call by a class (in its constructors) to the constructor in its parent. This would look like internal reuse, however it would often happen without any intervention by the programmer. As we want to understand what decisions programmers make with respect to inheritance, we separate out these calls. Since calls via `super` are somewhat under the control of programmers, we distinguish those from calls to the default constructor with the super attribute.

Note that we take a very liberal view of when reuse occurs. Whether external reuse or internal reuse, we will mark inheritance relationships (edges) as such if only one member is used. This means that a child class might inherit 100 methods from its parent, and only 1 of those 100 methods might be used, but we would still consider this an indication of reuse.

```
package org.jgraph.graph;
...
import javax.swing.TransferHandler;
...
public class GraphTransferHandler
    extends TransferHandler {
  ...
}
```

Fig. 7. The "framework" problem. Uses of some inheritance relationships may not be visible in the analysed code.

```
List list = new Vector();// Raw type
T aT = new R();          // R is a subtype of T
list.add(at);            // Subtype use of edge R to Object
S anS = (S)list.get(0);  // Cast from Object to S
                         // Only succeeds if R is a subtype
                         // of S
```

Fig. 8. Example of a how a "generic" container based on `java.lang.Object` relies on the subtype relationship

As noted above, we do not include edges to the standard API and third-party libraries. The main reason for this is that we do not have reliable information regarding which version of the API or libraries are assumed. Even if we had this information, including them would significantly increase the amount of code to be analysed (the standard API is one of the largest systems in the corpus), and the memory limitations of our tool would prevent us from doing the analysis.

4.3 Analysis Challenges

Because we do not consider the use of third-party libraries or the Java Standard API, the purpose of some inheritance relationships cannot be determined. For example, consider Figure 7 showing the `jgraph` class `GraphTransferHandler`. This class inherits from a class in the Swing framework and so where it is substituted for that class, `TransferHandler`, may only be visible in the Swing implementation, but not in the code we analyse. This also means that descendants of `GraphTransferHandler` might also be substituted for `TransferHandler`, but our analysis would not detect this. When we suspect this situation is possible, the edge will get the "framework" attribute.

Another limitation of our analysis is when generic types are implemented using casting to and from `java.lang.Object`. An object of one type can be put into a generic container and then cast to a different type on its removal. This behaviour depends on

```
class P {
  public void parent() { ... }
}
class C extends P {
  public void child() { ... }
}
class U {
  void user() {
    C aC = new C();
    aC.parent();
  }
}
```

Fig. 9. In some cases, the indicated invocation will show `P` as the invoked type, not `C`

a subtype relationship existing between the two types, as shown in figure 8. When we suspect this situation is possible, the edge will get the generic attribute.

Bytecode does not always directly map on to source code, and this can effect some of our analysis. One example is shown in Figure 9. Method invocation is indicated in the bytecode with an instruction such as `INVOKEVIRTUAL`, and will include information on which method is being invoked. In the case of the example, when `U` is compiled, we might expect that it is `C#parent()` that is written into the bytecode as the method being invoked, as that is a correct representation of what should happen. However, some compilers will write `P#parent` into the bytecode. As this information is not used in method lookup, the difference does not cause problems in execution, but it does affect model fidelity. It will mean it is possible for some relationships for which *external reuse* use occurs in the source code to to not be attributed as such in the model. We have been unable to reproduce this case ourselves with compilers we have access to, but do know it happens, albeit rarely, in code in the corpus.

5 Results

We present our results here organised around our research questions. Due to the volume involved, we cannot present all the data[1], so we present those results that best indicate trends, and of systems of interest such as `ant`, `freecol`, and `jre` (the largest system studied by any measure of size).

5.1 RQ1: Late-Bound Self-reference

Downcall edges must be CC edges. Figure 10 shows, for each system, the proportion of CC edges that are downcall edges. The systems are in increasing order of the number of CC edges. The order was chosen to determine whether or not there was a trend with

[1] Available from `http://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance`

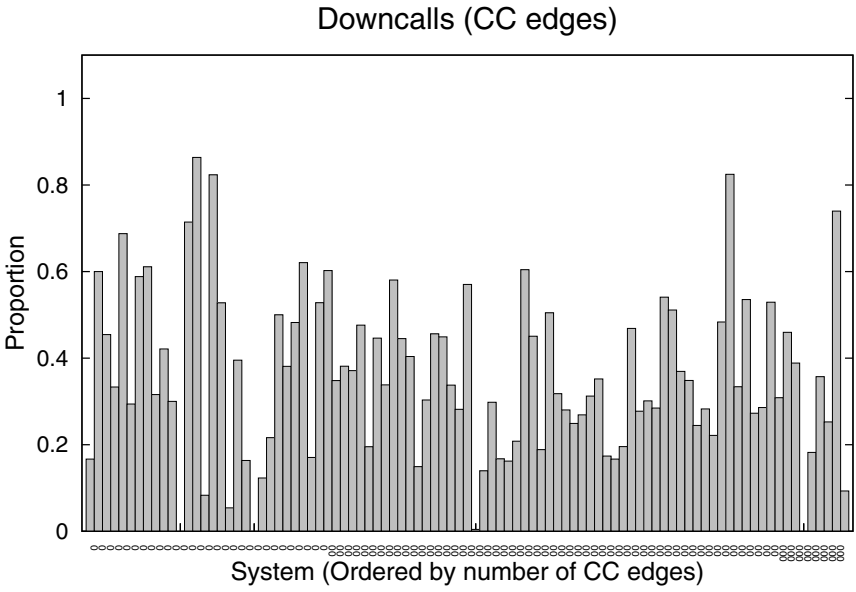


Fig. 10. Proportion of CC edges over which downcalls may occur. X-axis labels indicate order of magnitude of number of CC edges.

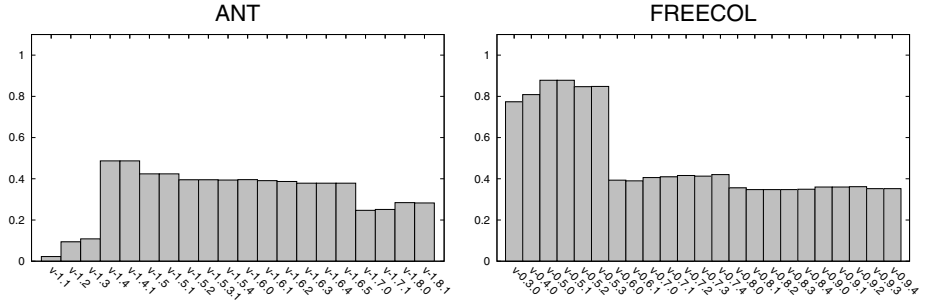


Fig. 11. Downcalls `ant` and `freecol` (x-axis is release order))

respect to this size metric (and the equivalent order will be used for most other charts). The x-axis labels indicate the order of magnitude of the number of edges (e.g. "00" indicates from 10^2 up to (not including) 10^3 edges).

The results indicate quite wide variation between systems, from zero (`freecs` — 61 edges, `jasml` — 21, `megamek` — 1283) to to 86% (`jFin_DateMath` — 22). The median is 34% (`aoi`). For the systems of interest, `ant` had 28%, `freecol` had 35%, and `jre` had 9%.

Figure 11 shows the downcall proportions for the releases of `ant` and `freecol`. Both show non-trivial uses of downcall (20% for `ant` and nearly 40% for `freecol`),

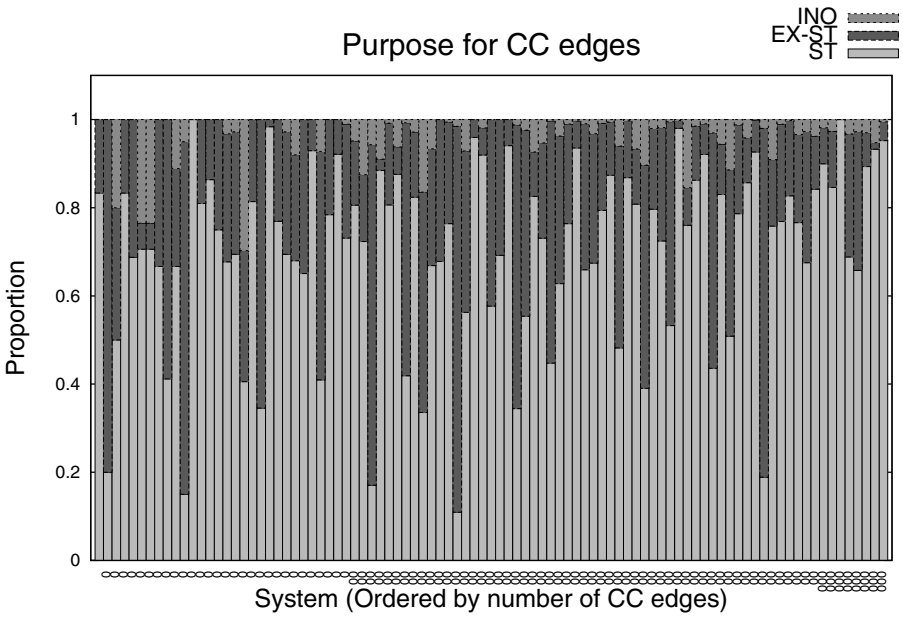


Fig. 12. This shows CC edges that are subtype edges (ST), external reuse edges but not subtype edges (EX-ST), or only internal reuse edges (INO)

but both also show quite large changes between some releases (increasing and decreasing).

There is no obvious trend with respect to size as measured by number of CC edges between user-defined types. Our conclusion is that late-bound self-reference plays a significant role in the systems we studied — around a third (median 34%) of CC edges involve downcalls.

5.2 RQ2: Subtype Relationship

CC, CI, and II edges can all be subtype edges. We present the results for each kind of edge separately. For the CC edges, we first identified those CC edges that were at least one of subtype, external reuse, or internal reuse. Figure 12 shows (bottom segment, "ST") the proportion of those CC edges that are subtype edges (other values will be discussed below), with the systems ordered as in Figure 10.

Again we see wide variation, with the smallest being at 11% (`checkstyle`), two with 100% (`magemek` with 1283 CC edges, `jasml` with 21), and the median at 76% (`jmeter`). The largest system (`jre`) had a measurement of 95%, indicating that almost all of the `extends` relationships between classes had some subtype use within its implementation.

Figure 13 shows the subtype use of all CI edges in a system. The bottom segment ("ST") indicates proportion of CI edges for which subtype use was seen. The second segment ("SUS") indicate the proportion of edges for which we suspect there is subtype

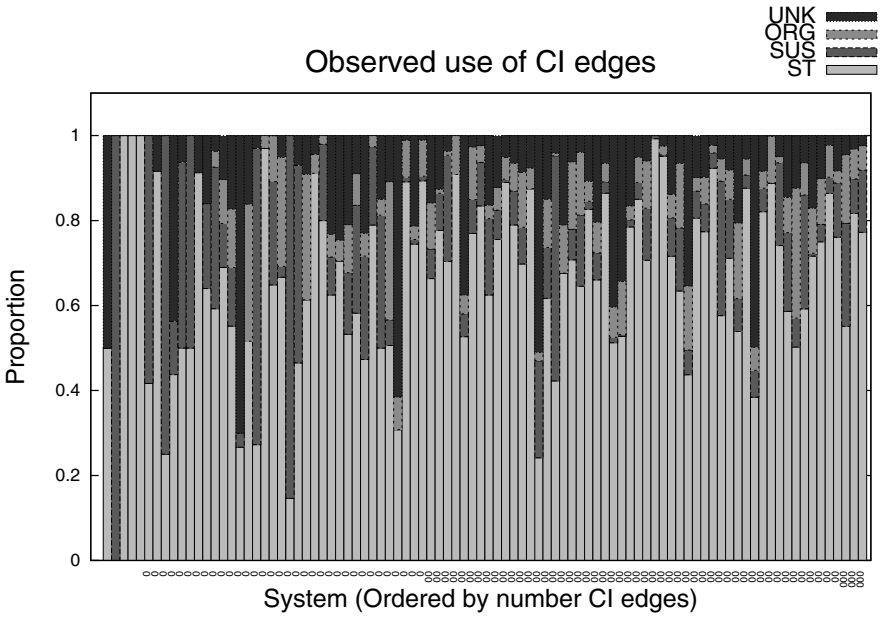


Fig. 13. CI edges that are subtype edges (ST), suspected to be subtype (SUS), organisational (ORG), or of unknown purpose (UNK)

use, but limitations of our analysis means we did not directly observe such. We include these so as to not bias our results against use of subtype. The other segments will be discussed below.

There is one system (`fitjava`) with no CI edges. Of the remainder, there are 3 systems (`nekohtml`, `jsXe`, and `joggplayer`) for which all CI edges were subtype edges, however they all had fewer than 10 CI edges. In a further 4 systems (`jasml`, `jmoney`, `jparse`, `javacc`) all CI edges where either subtype or suspected of being subtype edges. The median use was 69% (`checkstyle` considering subtype only, or 85% (`megamek` if including suspected subtype edges). For the other systems of interest, `jre` had 82% being subtype edges, or 91% including those suspected, `ant` had 63% and 78% respectively, with `freecol` having 83% and 94%.

Figure 14 shows the use of II edges, with the systems ordered in increasing order of number of II edges. There are 23 systems with no II edges (and so have no values in the chart), and 51 systems in total with fewer than 10 edges. As before, the bottom ("ST") segment shows the subtype edges. Of these, 13 systems had all II edges being subtype edges, however the largest (`jhotdraw`) had only 14 edges. The median use was 63% (`findbugs`). Of the systems of interest, `jre` had 71% (of 799 II edges), `ant` had 94% (18), and `freecol` had 67% (3). The system with the second largest number if II edges (`nakedobjects` with 349) had 66%.

Our conclusion is that at least two thirds of all inheritance edges are used as subtypes in the program — inheritance for subtyping is not as rare as Taivalsaari implies [25].

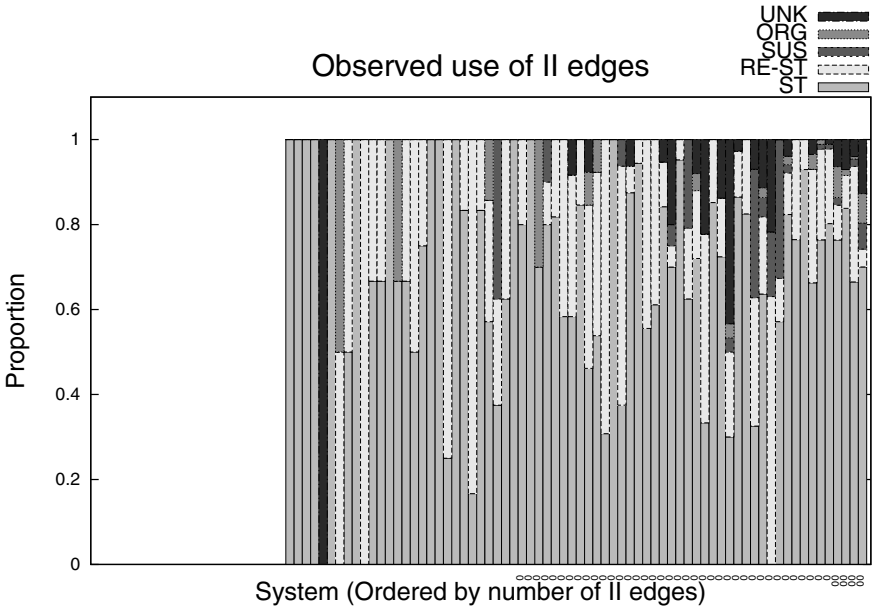


Fig. 14. II edges that are subtype edges (ST), reuse (RE-ST), suspected subtype (SUS), organisational (ORG), or unknown purpose (UNK). Systems with no II edges have no values (at left end)

5.3 RQ3: Inheritance vs. Composition

Our interest here is identifying use of inheritance that could have been changed to composition using a procedure such as that proposed by Bloch [2]. Briefly, the procedure is, remove the inheritance relationship and add to the child: a) a field that stores an instance of the parent class; and b) wrapper methods that delegate all references to attributes of original parent class to the instance stored in the field. This procedure would not apply when the purpose for using inheritance was to establish a subtype relationship, so we need to identify those edges that are internal or external reuse, but not subtype.

The procedure described by Bloch can be tedious to apply for parents with many members, and so it could be argued it is not always practical, however it is much simpler (and hence requires less effort) when only internal reuse is involved. Consequently we identify also those edges that are internal reuse only.

Figure 12 shows, as well as the proportion of subtype edges, those edges that are external reuse (and possibly internal reuse) but not subtype, and those edges that are internal reuse only.

The system with the largest proportion of external reuse (but not subtype) edges was `checkstyle`, with 88% of 193 CC edges. The median was 22% (`javacc`, 88 edges). For internal reuse edges only, the largest was 30% (`jpf`, 37 edges) and the median was 2% (`lucene`, 446 edges). There were 24 systems with no internal reuse only edges, the largest being `megamek`. For other systems of interest, `ant` had 20% external reuse

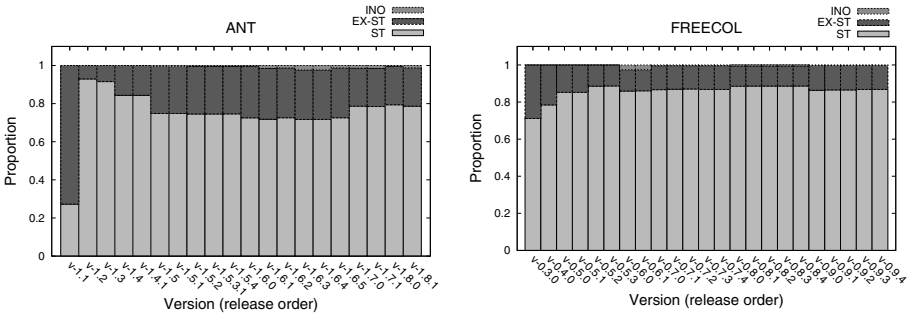


Fig. 15. CC edges that are subtype edges, external reuse edges but not subtype edges (EX-ST), or only internal reuse edges (INO) over time for `ant` and `freecol`

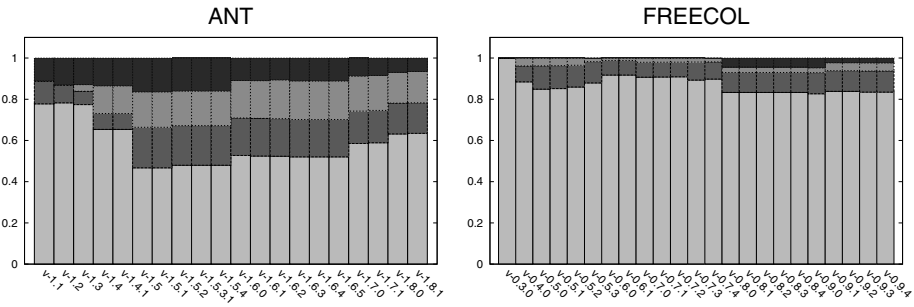


Fig. 16. Use of CI edges for `ant` and `freecol` (legend as for Figure 13, x-axis is release order)

and 1% internal reuse, `freecol` had 13% external and less than 0.5% internal, and `jre` had 4% external and less than 0.5% internal.

Figure 15 shows the same data as Figure 12 for `ant` and `freecol`. One point to keep in mind when interpreting these charts is that `ant` grew from 44 CC edges to 672 edges, and `freecol` grew from 53 to 392 edges. The proportions are remarkably constant after about the 5th release (`ant-1.5` has 401 CC edges, `freecol-0.6.0` has 239). It would be interesting to know why this is so, for example is it due to its architecture, or some other reason.

Figure 16 shows the same data as Figure 13 for `ant` and `freecol`. The subtype use is noticeably less for `ant` than `freecol`, although the subtype use increases over time. The subtype use in `freecol` for CI edges is similar to that for CC edges. The differences between the two systems could be indicative of the different nature of the systems.

Our conclusion is that there is generally opportunity for replacing inheritance with composition, with 22% or more uses of inheritance between classes needed for external reuse but not subtyping in half the systems we examined. For internal reuse edges only, there are many fewer opportunities for replacing inheritance with composition, but they do exist for 2% or more of such uses in half the systems. We cannot say whether

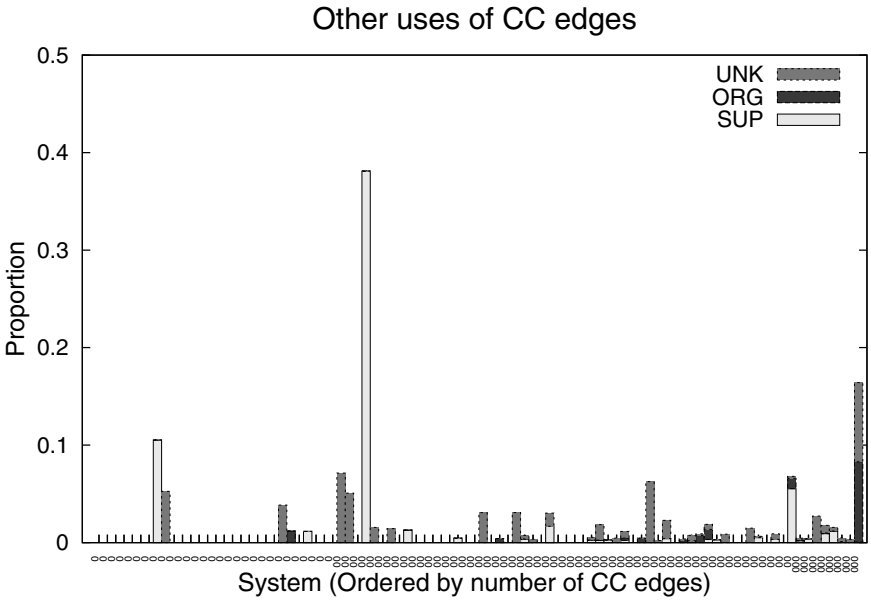


Fig. 17. Uses of CC edges that are not subtype, external reuse, or internal reuse edges — super constructor (SUP), organisational (ORG), or unknown purpose (UNK)

replacing inheritance with composition is worth the effort because we have no way to quantify the costs of not doing so. We do believe, however, that the prevalence of this use of inheritance is high enough to justify further research effort needed to understand how to quantify the costs, and also to give greater emphasis in teaching to avoid such uses of inheritance.

5.4 RQ4: Other Uses of Inheritance

It is instructive to consider those inheritance relationships that do *not* support at least one of external reuse, internal reuse, or subtype, as this helps us understand to what other purpose developers might use inheritance. For the remainder of this section we will only be referring to these as yet uncategorised edges.

We noted the possible use of interfaces (or classes) solely to define *constants*, and the use of marker interfaces. For the former, only 13 systems had CC edges where the parents held constants and 5 of these had more than 1%, the largest of these being `fitlibraryforfitnesse`, with 13% of 259 edges. For CI edges 45 systems had no occurrences, and 18 had more than 10%. The largest was 100% by `jasml` (but only 2 edges), but 4 systems had more than 50%. While some of these results were clearly due to such things as parser-generators (or similar), they do indicate that this idiom is fairly common, remembering that edges reported as subtype, external reuse, or internal reuse can also support this idiom.

For use of *marker* interfaces, 61 systems had no occurrences of CI edges being relationships to marker interfaces, however of those that did, they predominantly came from the larger systems. The largest proportion was 47% (`jext` with 43 CI edges), but (for example) `weka` had 14% of 1047 edges in this category. These results suggest that the use of marker interfaces is also a common (sole) reason for using inheritance.

As discussed in Section 4.3, some edges may be *framework* or *generic* edges. For CC edges, 58 systems had no occurrences of framework or generic edges that were also not constants or marker edges. Of those that did, 16 had less than 1%. The largest was 17% (`webmail`, with 24 CC edges). For CI edges, 38 had none, 8 had more than 10%, with `oscache` having 58% (of 12 edges). For II edges, only `jmeter` had any (5% of 20).

Of the remaining CC edges, one pattern we noticed was children whose only use of the relationship with the parent class was via calling a non-default constructor (via `super`). Another pattern we noticed for all edge types was one edge may appear to have no purpose, but an edge from a sibling to the common parent was one of subtype, internal reuse, or external reuse. It could be that the parent was playing an organisation role, indicating types that are conceptually related but which relationship may play no role within the implementation. We report such edges as *organisational* ("ORG" in figures 13, 16, and 14).

Figure 17 shows the unused CC edges with (light gray) showing those whose sole use is super constructors, (medium gray) showing remaining edges that were organisational, and edges with no purpose was identified in our analysis in (dark). As can be seen, the measurements are mostly small. The large value (38%) for calls to super constructors is in `trove`, and are by classes that appear to be all generated. There are 57 systems where we could associate some purpose to all edges, with 15 having more than 1% and being mostly systems with a large number of CC edges. The largest is `jre` with 8% of CC edges having no obvious purpose.

Figure 13 also shows our analysis of those CI edges that are not subtype edges. The third segment ("ORG") show those CI edges that are organisational, and the top segment ("UNK") show the proportion of CI edges for which we could find no purpose. Only 9 systems had no edges for which we could find no purpose (`jparse` was the largest of these having 41 CI edges). The system with the most such edges was `jre` (470 edges, 9% of its CI edges), with the median number of edges being 20 (`velocity`, 32%). The system with the largest proportion of such edges was `c_jdbc` (70% of 30 edges), and the median proportion was 15% (of the 113 CI edges of `pooka`).

Figure 14 also shows the other uses we observed of II edges. The segments indicate proportions for the different categories as in Figure 13, with the addition of a (second) segment ("RE-ST") are "reuse" edges that are not subtype. In the context of II edges this means that a method was seen to be invoked on an interface type, but in fact that edge was declared in an ancestor interface. While the majority are subtype, the reuse category stands out. This category shows interfaces whose sole purpose is to indicate where there is shared behaviour between types in the implementation. There are 54 systems with such edges. The maximum value was 100% (`colt`, 3 edges). Of the 70 systems with any II edges, the median value was 17% (`jgrapht`, 6 edges). The largest

system `jre` had 4% (of 799 edges), `ant` had 6% (18 edges) and `nakedobjects` had 27% of 349 edges.

For edges with no purpose, this was true of all 2 of `checkstyle`'s II edges and 13% of `jre`'s edges. Only 21 systems had any such edges.

Our conclusion is that our conservative inheritance model classified over 58536 out of 67529 (87%) of all edges in our graph (38122/39973 or 95% of all CC relationships) — as either subtype edges, external reuse, and internal reuse, that is, subtype, external, and internal reuse explain most of the inheritance relationships in our corpus.

6 Discussion

As indicated, there are some limitations to our analysis, so the natural question is to what degree do they threaten the validity of our conclusions. The first point to make is that the results for RQ2, and RQ3 indicate edges for which we actually observed subtyping and reuse respectively. That is, we may have false negatives, but no false positives, so the analysis is conservative with respect to these questions.

Regarding RQ1, it is possible our results overstate reality, since we assume that if a self-call exists, and a descendant overrides that method, that a downcall will occur at the self-call. Since this depends on the run-time behaviour of the system, we cannot be sure that this will always be the case. We did limited manual inspection and found no overestimation.

For RQ4, we are quite confident about the accuracy of the measurements for classes and interfaces containing only constants, and for use of marker interfaces. For those edges that we reported for framework and generic use, if our assumption is correct and they are in fact used within frameworks or for generic types, then this would indicate the degree to which our subtype results are under-reporting the true situation. We know this is the case for a number of systems through manual inspection.

The most uncertainty exists for those edges we cannot easily classify, those that we report as UNK. The CI category of edges had the highest incidence of such edges (figure 13), which is surprising, given there is no obvious purpose to a class implementing an interface other than use the subtype relationship. Only 9 systems had no edges in this category (`jparse` was the largest of these having 41 CI edges). The system with the most such edges was `jre` (470 edges, 9% of its CI edges), with the median number of edges being 20 (`velocity`, 32%). The system with the largest proportion of such edges was `c_jdbc` (70% of 30 edges), and the median proportion was 15% (`pooka`, 113 CI).

We have manually examined a number of unused CI edges. We classified them as one of: implementations provided by the system for use by other clients; intended for future development; unable to classify; and having no good reason for existing.

For the first category we noted that the systems were frameworks or libraries, there were no names declared of the relevant types, the names had "Default" or "Adaptor" in them (e.g. `DefaultActionNameBuilder` in `struts`), or we had some other reason (e.g. comments) to think the types were intended for clients of the framework, not the framework itself. The `jre` is a good example of this. Given its purpose, it should be unsurprising that some of the types it provides are not used within it, and

so if those types are defined using inheritance we cannot expect to see the use of the inheritance relationship in any way within `jre`. In fact, we observed subtype use of 95% of `extends` edges in `jre`.

This represents the other side of the "framework" issue we discussed in section 4. Because we do not include third-party libraries in our analysis, we cannot detect uses of inheritance that cross the boundary between system code and third-party code.

The last three categories were ever more subjective. The category "future development" was adjudged if we had some reason (e.g. comments) to believe the relationships would be used in future releases. The category "unable to classify" we choose if we had some reason to not be able to make a judgement. For example, the possibility of reflection meant it was possible we missed some cases (although many would in fact be correctly identified by our tools), or the complexity of the design and the limited time available meant we could not come to any conclusion. The last category (no good reason) we chose when we could find no reason for the relationship (e.g. when we found no declarations of the interface type but many declarations with the implementation types). While this discussion refers specifically to CI edges, we performed a similar analysis for a subtype of CC and II edges as well.

Despite the subjectivity of some of our analysis, we did see cases where even with the most generous interpretation there seemed no reason for having an inheritance relationship. Nevertheless, such occurrences were fairly rare, and so we feel we can quite confidently state that there is little evidence of systematic unnecessary inheritance.

There is quite wide variation in the size of the systems. Using the total of CC, CI, and II edges as a size metric (see also Table 1), `jre` was the largest (11636), followed by `jruby` (4418), and `drjava` (2474). Most systems were only 10% of `jre` (75 with fewer than 1000 edges, the smallest `jsXe` had 14). Despite this, no one system dominates any of our measurements, suggesting that how inheritance is used is not determined by the size of the system.

The longitudinal studies (figures 11, 16, and 15) show some abrupt changes. They all correspond to significant changes in the number of inheritance relationships (CC, CI, and CC respectively), and generally to significant changes in the overall code bases. This all points to changes in the design, however our measurements cannot show what led to those changes. That will require much deeper analysis, both of the code base and of the developers' thinking.

We have only considered relationships between system types, and so framework relationships (e.g. Figure 7) are not modelled. Due to the framework issue, we believed we would not be able to adequately represent the situation, and indeed when we include such edges we see generally a lower proportion of subtype edges. Nevertheless, they do indicate decisions made by the developer and we would like to be able to study these edges in more detail in the future.

We take a very liberal view of when we classified a relationship as for subtype or reuse. For example, there may be only one point in the implementation where subtype is needed, but many uses of external reuse, however we would report that edge as use of subtype. Our view is coarse-grained, but it does give an overall indication of how inheritance is used. With these results, we can now identify more specific questions of how inheritance is used, and they also help us restrict the systems we need to investigate

to answer the questions. In particular, it would be valuable to revisit the studies done previously ([10,6,14]) but using (for example) degree of use of subtype versus external reuse as the independent variable, rather than DIT. Such studies would determine the validity of advice regarding composition versus inheritance.

Also, we report use of subtyping from a static perspective, asking essentially does the code pass the type checks. There is still the possibility that, while there may be a subtype relationship, this relationship does not correspond to conceptual specialisation. In fact, recent work by Pradel and Gross on detecting subclasses that are unsafe substitutes for their superclasses suggests this may be an issue [21]. Their technique may allow a more careful study of developers' use of subtype.

While authors such as Meyer [20] and Taivalsaari [25] suggest there are many uses for inheritance, our studies suggests there are only two main uses in Java code. This could be due to how we classify relationships. More study is needed in this regard.

Our results also appear to disagree with Taivalsaari's observations. He characterised use of subtype as rare. It would be interesting to know whether this is due to the languages he referred to (Smalltalk and C++), due to the reasons programmers use inheritance having changed over time, or due to his observations being based only on his experience. Only further objective empirical studies will definitively answer such questions.

7 Conclusions

Our overall goal is to understand how design decisions impact the quality (however it is defined) of software. We are currently examining how inheritance is used by developers. In this paper we have presented a study of 93 open-source Java systems. We found about one third of subclasses rely on late bound self-reference (downcalls) to customise their superclasses' behaviour (RQ1). Java developers mostly use inheritance for subtyping, with about two thirds of inheritance relationships needed for this (RQ2). While there is not a large opportunity to replace inheritance with composition (RQ3), the opportunity is significant (median of 2% of uses are only internal reuse, and a further 22% are only external or internal reuse). While there are other uses of inheritance, their use is not generally significant (RQ4).

Our results suggest there is no need for concern regarding abuse of inheritance (at least in open-source Java software), but they do highlight the question regarding use of composition versus inheritance. If there are significant costs associated with using inheritance when composition could be used, then our results suggest there is some cause for concern. We believe understanding these costs is an important open question.

This research also provides support for our previous work [28]. Our conclusion in that work was that there was a considerable amount of use of inheritance. We can now say that most of that use is justified, in that without it code would not compile (at minimum). Whether this use was the best design choice remains to be seen, but it emphases the importance of inheritance, at least for Java programmers. It also means that future research on the use of inheritance can use the same systems we have used without concern that unnecessary use of inheritance might taint the results.

There are many other possible avenues of research following from the work presented here. One of particular interest to us is to understand the rationale behind a

programmer's use of inheritance. We will use qualitative techniques based on grounded theory to gain this understanding. It is also important that our research be replicated, including with other Java (particularly closed-source) systems, and systems in other languages.

References

1. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22(10), 751–761 (1996)
2. Bloch, J.: Effective Java, 2nd edn. Addison-Wesley (2008)
3. Briand, L.C., Daly, J., Porter, V., Wüst, J.K.: A comprehensive empirical validation of design measures for object-oriented systems. In: METRICS 1998: Proceedings of the 5th International Symposium on Software Metrics, pp. 246–257. IEEE Computer Society, Washington, DC (1998)
4. Budd, T.: An Introduction to Object-Oriented Programming, 3rd edn. Addison-Wesley (2002)
5. Cargill, T.A.: The case against multiple inheritance in C++. USENIX Computing Systems 4(1), 69–82 (1991)
6. Cartwright, M.: An empirical view of inheritance. Information and Software Technology 40, 795–799 (1998)
7. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20(6), 476–493 (1994)
8. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object oriented design. In: OOPSLA, pp. 197–211 (1991)
9. Cook, W.R., Hill, W., Canning, P.S.: Inheritance is not subtyping. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1990, pp. 125–135. ACM, New York (1990)
10. Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M.: Evaluating inheritance depth on the maintainability of object-oriented software. Empirical Software Engineering 1(2), 109–132 (1996)
11. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. PWS Publishing Co., Boston (1998)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley Publishing Company, Reading (1994)
13. Gil, J(Y.), Maman, I.: Micro patterns in Java code. In: OOPSLA, pp. 97–116 (2005)
14. Harrison, R., Counsell, S., Nithi, R.: Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. Journal of Systems and Software 52, 173–179 (2000)
15. Holub, A.: Why extends is evil: Improve your code by replacing concrete base classes with interfaces (August 2003), `http://JavaWorld.com`
16. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming (June/July 1988)
17. Liskov, B.: Keynote address — data abstraction and hierarchy. SIGPLAN Notices 23(5), 17–34 (1987)
18. Manel, D., Havanas, W.: A study of the impact of C++ on software maintenance. In: International Conference on Software Maintenance, pp. 63–69 (1990)
19. Meyer, B.: Genericity versus inheritance. In: OOPSLA, pp. 391–405. ACM, New York (1986)

20. Meyer, B.: The many faces of inheritance: a taxonomy of taxonomy. IEEE Computer 29(5), 105–108 (1996)
21. Pradel, M., Gross, T.R.: Automatic testing of sequential and concurrent substitutability. In: International Conference on Software Engineering, ICSE (2013)
22. Schmager, F., Cameron, N., Noble, J.: GoHotDraw: evaluating the go programming language with design patterns. In: Evaluation and Usability of Programming Languages and Tools, PLATEAU 2010, pp. 10:1–10:6. ACM, New York (2010)
23. Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In: OOPSLA, pp. 38–45 (1986)
24. Sumption, B.: Inheritance is evil, and must be destroyed (2007), `http://berniesumption.com/software/inheritance-is-evil-and-must-be-destroyed` (last accessed December 2012)
25. Taivalsaari, A.: On the notion of inheritance. Comp. Surv. 28(3), 438–479 (1996)
26. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: A curated collection of Java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference (APSEC 2010), pp. 336–345 (December 2010), `www.qualitascorpus.com`
27. Tempero, E., Counsell, S., Noble, J.: An empirical study of overriding in open source Java. In: Thirty-Third Australasian Computer Science Conference (ACSC 2010), Brisbane, Australia. The Conferences in Research and Practice in Information Technology (CRPIT) Series, vol. 102, pp. 3–12 (January 2010)
28. Tempero, E., Noble, J., Melton, H.: How do Java programs use inheritance? An empirical study of inheritance in Java software. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 667–691. Springer, Heidelberg (2008)
29. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: CASCON, pp. 125–135 (1999)
30. Waldo, J.: Controversy: The case for multiple inheritance in C++. USENIX Computing Systems 4(2), 157–172 (1991)

Is This a Bug or an Obsolete Test?

Dan Hao[1], Tian Lan[1], Hongyu Zhang[2], Chao Guo[1], and Lu Zhang[1]

[1] Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, School of Electronics Engineering and Computer Science
Peking University, Beijing, 100871, P.R. China
[2] Tsinghua University, 100084, P.R. China
{haod,lantian12,guochao09,zhanglu}@sei.pku.edu.cn, hongyu@tsinghua.edu.cn

Abstract. In software evolution, developers typically need to identify whether the failure of a test is due to a bug in the source code under test or the obsoleteness of the test code when they execute a test suite. Only after finding the cause of a failure can developers determine whether to fix the bug or repair the obsolete test. Researchers have proposed several techniques to automate test repair. However, test-repair techniques typically assume that test failures are always due to obsolete tests. Thus, such techniques may not be applicable in real world software evolution when developers do not know whether the failure is due to a bug or an obsolete test. To know whether the cause of a test failure lies in the source code under test or in the test code, we view this problem as a classification problem and propose an automatic approach based on machine learning. Specifically, we target Java software using the JUnit testing framework and collect a set of features that may be related to failures of tests. Using this set of features, we adopt the Best-first Decision Tree Learning algorithm to train a classifier with some existing regression test failures as training instances. Then, we use the classifier to classify future failed tests. Furthermore, we evaluated our approach using two Java programs in three scenarios (within the same version, within different versions of a program, and between different programs), and found that our approach can effectively classify the causes of failed tests.

1 Introduction

After software is released to its user, it is still necessary for developers to modify the released software due to enhancement, adaptation or bug[1] fixing [32], which is typically referred to as software evolution. As estimated, 50%-90% of total software development costs [31,46,16] are due to software evolution.

In software evolution, developers usually perform regression testing to make sure that their modifications to the software work as expected and do not introduce new faults. Modifications in software evolution include changing functionality, fixing bugs, refactoring, restructuring code, and so on.

In software evolution, some modifications (e.g., changing functionality) may further imply changes of the specifications on program behaviors. In such cases,

[1] Bugs and faults are used interchangeably in this paper.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 602–628, 2013.

developers may need to modify the corresponding tests to reflect developers' changing expectation on program behaviors. Thus, when a regression test fails[2], it may just indicate that the specification the test represents is obsolete and the test itself needs repair. If developers do not want to change their specifications of the program in software evolution, a regression test failure may indicate that developers should modify the program without modifying the test. In other words, the tests and the software under test should be developed and maintained synchronously [59].

Specifically, developers for software systems often reuse and adapt existing tests to evolve test suites [40]. As these existing tests are developed for the old version of the software, when some of them cause failures of the new version in regression testing, the failures may not be due to bugs introduced in the modifications [36,8]. That is to say, some failures are due to bugs in the modified source code under test, but other failures are due to the obsoleteness[3] of some tests. Specifically, an internal study of ABB Corporate Research indicates that around 80% of failures in regression testing are due to bugs in the software under test and the other failures are due to obsolete tests [45].

As testing frameworks like JUnit[4] have been widely used in practice, the tests constructed by developers are also pieces of code. For example, Fig. 1 presents a Java program and its tests. To distinguish the tests and software under test, we denote the source code of the tests and the source code of the software under test as *test code* and *product code* to make our presentation concise. In practice, it is necessary for developers to identify whether such a failure is caused by a bug in the source code of the software under test or an obsolete test. Otherwise, developers would not know how the regression test failures reflect the quality of the software under test. However, as it may be challenging to guess developers' intention as the product code does not reflect developers' intention(i.e., whether or not changing the specifications of the program) explicitly in software evolution, it is not straightforward to know whether the failure of a regression test is due to faulty program changes or obsolete specifications represented by tests. Moreover, as reported in a technical report from Microsoft Research, the test code is often larger than the product code in many projects [50]. As both the test code and the product code are large, it is tedious and difficult for developers to determine the cause of a failure by manually examining the test code and/or the product code. Furthermore, if developers take obsolete tests as bugs in the software, they may submit some false bug reports and thus incur extra burdens for bug-report processing (e.g., triaging [54]).

Furthermore, the knowledge about whether regression test failures are due to bugs or obsolete tests not only can help understand what is going on in the

[2] If the output of a test is as expected (i.e., being as asserted), we call that the test passes; otherwise, we call that the test fails.

[3] Due to the difference between the new version and the old version of software under test, some existing tests for the old version cannot be used to test the new version. We call tests that need modification for the new version as obsolete tests.

[4] `http://www.junit.org`

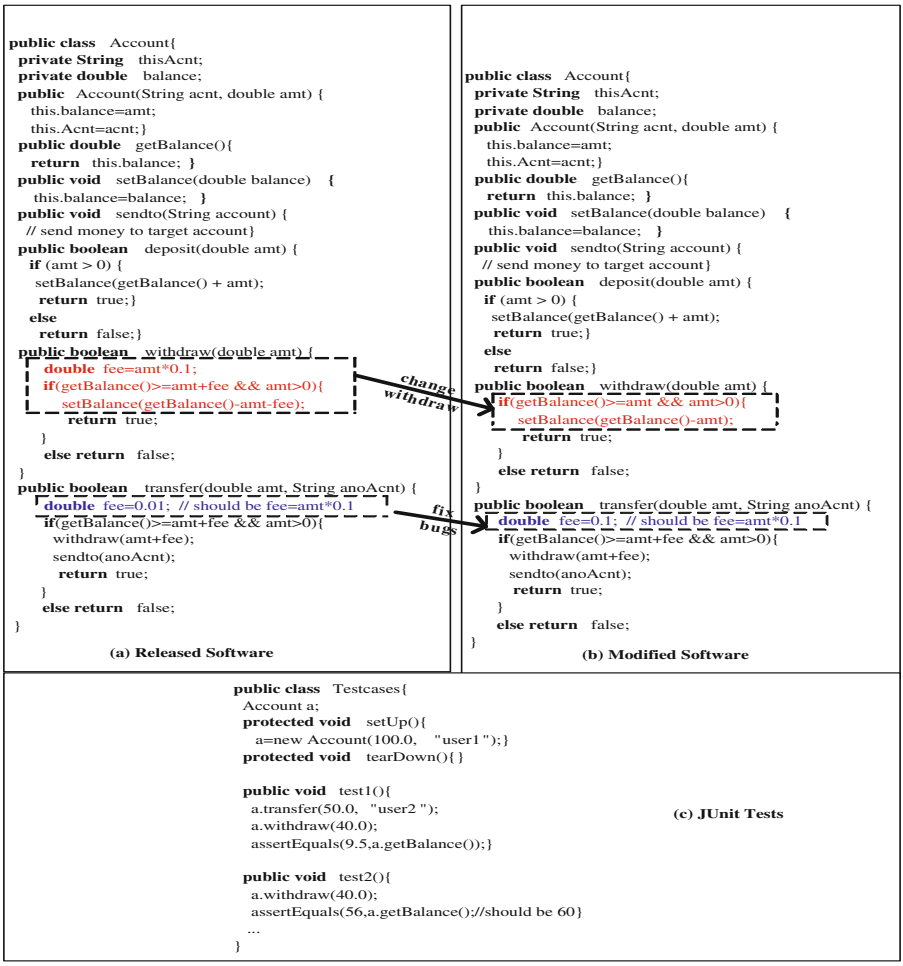


Fig. 1. An Example Program and Its JUnit Tests

regression testing process but also can help reduce the cost of code modification to fix bugs or repair tests. If the failure is due to a bug, developers can use automated techniques (e.g., [52,56]) to decrease the cost of debugging in the product code; if the failure is due to an obsolete test, developers can also use automated techniques (e.g., [6,7,8]) to decrease the cost of test repair in the test code. It should be noted that techniques for automated debugging typically assume tests to be correct and focus on the product code to fix bugs. Similarly, techniques for automated test repair typically assume that the cause of the failure lies in the obsoleteness of tests. Therefore, it becomes the preceding condition for developers to identify the cause of the failure when they observe a failure [41]. That is to say, understanding the cause of a failure actually serves as an indicator

for whether to apply automated debugging techniques or automated test-repair techniques.

In this paper, we propose a novel approach to classifying the causes of regression test failures for Java programs using JUnit as the framework for regression testing. In particular, we transform the problem of classifying the cause (i.e., buggy product code or obsolete test code) of a regression test failure into a problem of learning a classifier based on the data of various features related to failures. Specifically, our approach adopts the Best-first Decision Tree Learning algorithm [47,48], which is one of the typical machine-learning algorithms in the literature. Moreover, we collect the data of the failure related features used for classification via analyzing the software under test and its test code.

To evaluate our machine-learning based approach, we performed three empirical studies on two Java programs: Jfreechart and Freecol. The first study aims to evaluate whether our approach is effective when being applied for the same version of a program. That is, the training set and the testing set consist of instances from the same version of a program. The second study aims to evaluate whether our approach is effective when being applied between versions of a program. That is, the training set and the testing set are instances of different versions of a program. The third study aims to evaluate whether our approach is effective when being applied between different programs. That is, the training set and the testing set are instances of different programs. According to the results of our empirical studies, our approach is effective in correctly classifying the causes of regression test failures when it is applied within the same program (including the same version and different versions).

In summary, this paper makes the following main contributions.

- First, we present a machine-learning based approach to classifying the causes of regression test failures. To our knowledge, this is the first piece of research that tries to classify the cause of a regression test failure as a bug in the product code or an obsolete test.
- Second, we performed three empirical studies to evaluate the effectiveness of the proposed approach in three scenarios: being applied within the same version, being applied within different versions of a program, and being applied between different programs.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 illustrates the problem in this paper by an example. Section 4 presents the details of our approach. Section 5 presents the setup of our empirical studies. Section 6 presents the findings of the empirical studies. Section 7 presents the discussion and Section 8 concludes.

2 Related Work

To our knowledge, the work presented in this paper is the first approach to classifying the causes of regression test failures in software evolution. The research most related to our work is fault repair, including debugging in the product code

and test repair, which will be discussed in Section 2.1 and Section 2.2. Our work is also related to regression testing as our work deals with tests in regression testing, and thus we will discuss regression testing techniques in Section 2.3. Furthermore, our work can be viewed as an application of machine-learning and thus we will discuss application of machine-learning in software quality engineering in Section 2.4.

2.1 Debugging in Product Code

Software debugging focuses on identifying the locations of faults and then fixing the faults by replacing the faulty code with the correct code.

Most existing research [2,22,52,64] on software debugging focuses on the first step, which is fault localization. Typically, spectrum-based fault-localization approaches [20,27,34] compare the execution information of failed tests and that of passed tests to calculate the suspiciousness of each structural units, and then localize the locations of faulty structural units by ranking the structural units based on their suspiciousness. As the effectiveness of these approaches is dependent on the test suites [1] and faults, some research focuses on improving these approaches via test selection [19] and generation [51]. Besides these spectrum-based fault localization techniques, some researchers transform fault localization to other mathematical problems, like the maximal satisfiability problem [28] and the linear programming problem [9]. To fix bugs, several techniques [17,56] have been proposed to automate patch generation. For example, Weimer et. al [56] proposed to fix faults by using genetic programming to generate a large number of variants of the program. However, as these techniques may generate nonsensical patches, Kim et al. [29] proposed a patch generation approach (i.e., PAR), which uses fix patterns learned from existing human-written patches.

The research on fault localization is related to our work because potentially these techniques may be extended to solve our problem. To verify the effectiveness of these techniques on classifying the causes of regression test failures, we have conducted a preliminary experiment and found that direct application of these fault-localization techniques can hardly correctly classify the causes of regression test failures. Details of this experiment are presented in Section 7.

2.2 Test Repair

When changing requirements invalidate existing tests, tests are broken. Besides deleting obsolete tests [69] or creating new tests to exercise the changes [57], test-repair techniques [58] are proposed to repair broken tests rather than removing or ignoring these tests. Specifically, Galli et al. [15] proposed a technique to partially order broken unit tests rather than arbitrary order, according to the sets of methods these tests called. Daniel et al. [8] presented a technique (i.e., ReAssert) based on dynamic analysis and static analysis to find repairs for broken unit tests by retaining their fault-revealability. As ReAssert cannot repair broken tests when they have complex control flows or operations on expected values, Daniel et al. [6,7] presented a novel test-repair technique based on symbolic execution to

improve ReAssert to repair more test failures and provide better repairs. These techniques aim to repair broken unit tests in general, while some techniques have been proposed to repair broken tests in graphical user interfaces [36] or web applications [24].

To learn whether existing test-repair techniques are applicable in real practice, Pinto et al. [41] conducted an empirical study on how test suites evolved and found that test repair does occur in practice.

Existing research on debugging in product code and test repair (including those introduced by Section 2.1 and Section 2.2) assumes that developers have known whether the failure to be due to the product code or the test code. That is, when a failure occurs, developers may have to manually determine the cause of this failure before applying existing techniques on debugging in the product code or on test repair. Without such knowledge, developers risk to locate the faults in the wrong places. Unfortunately, to our knowledge, except our work reported in this paper, there is no previous study in the literature on this issue.

2.3 Regression Testing

Regression testing [33,66] is a testing process, whose aim is to assure the quality of a program after modification. After a program is modified, developers often reuse existing tests for the program before modification and may add some tests for the modification. As it is time-consuming to run the aggregated tests, many test selection and/or reduction techniques [5,21,65,71] have been proposed to reduce the number of tests used in regression testing. To optimize the cost spent on regression testing, test prioritization techniques [61,62,68] have been proposed to schedule the execution order of tests. Most research in test selection, reduction and prioritization investigates various coverage criteria, including statement coverage, function coverage [12], modified condition/decision coverage [44], and so on. Other research investigates various test selection, reduction, and prioritization algorithms, including greedy algorithms [25], genetic algorithms [35], integer linear programming based algorithms [5,21,71], and so on.

Our work is related to regression testing, especially related to test selection. However, our work aims to determine the causes of regression test failures, whereas test selection aims to select tests that are effective in exposing faults in the modified program. Specifically, test selection aims to select tests whose output becomes obsolete for the new version, whereas our work tends to identify the tests that become obsolete and should be modified to test the new version.

2.4 Application of Machine Learning in Software Quality Engineering

It is a relatively new topic to apply machine learning to software quality engineering [4,14,18,23,42]. Brun and Ernst [4] isolated fault-revealing properties of a program by applying machine learning to a faulty program and its fixed version. Then the fault-revealing properties are used to identify other potential

faults. Francis et al. [14] proposed two new tree-based techniques to classify failing executions so that the failing executions resulting from the same cause are grouped together. Podgurski et al. [42] proposed to use supervised and unsupervised pattern classification and multivariate visualization to classify failing executions with the related cause. Bowring et al. [3] proposed to apply an active learning technique to classify software behavior based on execution data. Haran et al. [23] proposed to apply the Random Forest algorithm to classify passing executions from failing ones. Wang et al. [53] proposed to apply Bayesian Networks to predict the harmfulness of a code clone operation when developers' performing copy-and-paste operation. Host and Ostvold [26] proposed to identify the problem in method naming by using data mining techniques. Zhong et al. [70] proposed an API usage mining framework MAPO using clustering and association rule mining. Furthermore, machine-learning techniques have also been widely applied to software defect prediction [37,30,60].

Generally speaking, the existing research on application of machine learning in software quality engineering mostly aims to automate fault detection or identify failing executions, whereas our work aims to identify whether faults are in the product code or in the test code.

3 Motivating Example

In software evolution, if developers' intention (e.g., changing functionality) incurs specification changes when making modifications, such failures may indicate obsolete test code. If developers' intention does not incur specification changes when making modifications, such failures probably indicate faulty product code. As code changes do not explicitly reflect developers' intention, it is challenging to determine through automatic program analysis whether an observed failure in regression testing is due to bugs in the product code or obsolete tests.

Fig. 1 presents an example Java class *Account* (including the version before modification and the version after modification) and its JUnit tests. The former version of *Account* is shown by Fig. 1(a), which contains a bug in method $transfer$. Fig. 1(c) gives its JUnit tests, including two tests (at the test-method level) $test1$ and $test2$.

When developers run this version of *Account* with the two tests, $test1$ fails but $test2$ passes. To fix the bug in *Account* that causes the failure of $test1$, developers may modify method $transfer$, shown by Fig. 1(b). In using this version of class *Account*, the bank wants to remove extra fees when consumers withdraw their savings, and thus developers have to modify method *withdraw* of *Account*. In summary, when evolving *Account* from Fig. 1(a) to Fig. 1(b), developers modify two methods of *Account* due to different reasons. Method *withdraw* is modified because developers change their expectations on the behavior of *withdraw*, whereas method $transfer$ is modified because developers want to fix a bug in the method.

After modification, two failures are observed when developers run the modified software in Fig. 1(b) with $test1$ and $test2$. The failure for $test1$ is due to modified

Account, whereas the failure for *test*2 is due to the obsoleteness of this test. That is, the two failures have different causes (i.e., either in the product code or in the test code), resulting from developers' different intentions. As it is hard to automatically induce developers' intention based on only the software and its tests, it is not straightforward to tell whether an observed failure is due to the product code or the test code. That is, it is a challenging problem to classify the cause of an observed failure in software evolution. To our knowledge, our work is the first research that tries to solve this problem.

4 Approach

Despite the difficulty of our target problem, there are still some clues. For example, the complexity of the source code, the change information between versions, and testing information of the regression test failures may be related to the cause classification of a regression failure. Specifically, as developers are more likely to make mistakes in complex code, the failure for a regression test whose product code is complex is more likely due to bugs in the product code than the obsolete test. For example, as shown by Fig. 2, which shows the static call graphs of *test*1 and *test*2, the product code tested by *test*1 is complex as many methods of *Account* are called during the execution of *test*1, and thus the failure for *test*1 is probably due to bugs in the product code. Furthermore, more frequently changed and less tested product code is intuitively to be fault-prone. Since it is difficult to obtain some discriminative rules to classify the cause of an observed failure based on these clues, we present a machine-learning based technique to learn a classifier by using the failures whose causes are known.

In the following, we first give an overview of the proposed approach in Section 4.1. Then we present the features that we extract from the software under test and its test code in Section 4.2. Finally, we present how we train the classifier and use the classifier to classify regression test failures in Section 4.3.

4.1 Overview

In our approach, we view the problem of classifying the cause of a regression test failure as a problem of learning a classifier based on the data of failure-related features, which can be extracted and collected by analyzing the software under test and its test code. Specifically, we adopt a typical machine-learning algorithm, the Best-first Decision Tree Learning algorithm [47,48].

In existing software development environment, when a failure occurs in the execution of a test, there is no place to record whether the cause of the failure is due to the product code or the obsolete test. To collect failures for training our classifier, our approach requires developers to record the cause of each failure when they resolve a regression test failure. After labeling the cause of each failure, our approach assigns values to seven failure-related features by statically and dynamically analyzing the software under test and its test code. Using the failures with their causes and failure-related features, our approach trains a classifier. Finally, the trained classifier can be used to classify future failures.

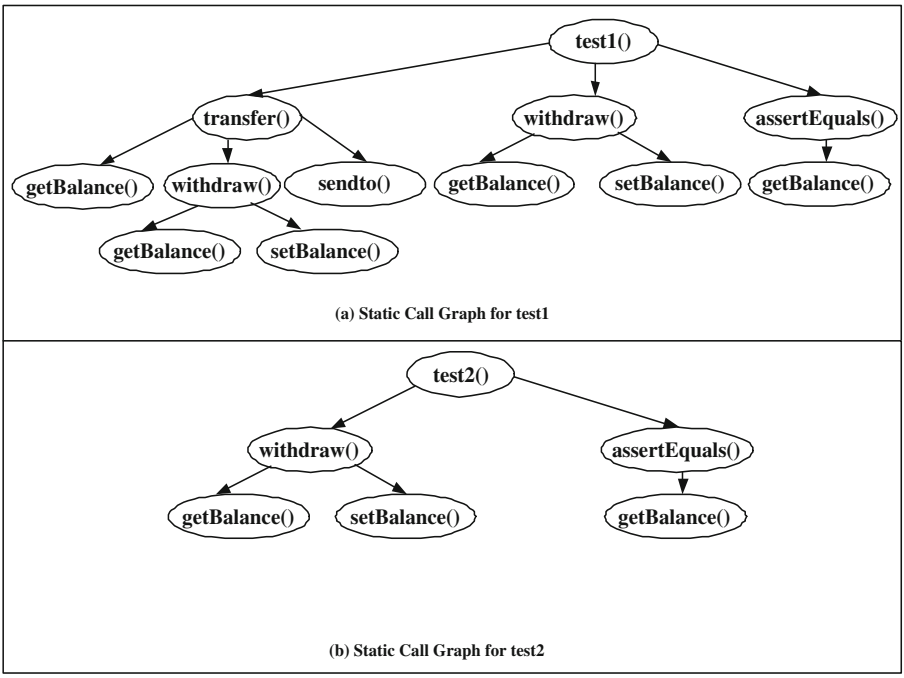


Fig. 2. Static Call Graph for the Tests

4.2 Identified Features

Our approach uses a set of features related to regression test failures to identify whether the cause of a regression test failure is due to the product code or the test code. Specifically, we use seven features, which can be further divided into three categories (i.e., two complexity features, one change feature, and four testing features).

Complexity Features. The complexity features are concerned with how complex the interaction between the test and the software under test is. Intuitively, the more complex the interaction is, the more likely the test can reveal a bug in the software under test. In other words, the more complex the interaction is, the more likely the failure is due to a bug in the product code.

As our target is Java programs using the JUnit testing framework, we are able to extract complexity features via statically analyzing the call graph of each test. As a JUnit test is a piece of executable source code containing testing content (i.e., a sequence of method invocations), we can determine what methods each JUnit test invokes and use such information to construct static call graphs for JUnit tests.

Given a JUnit test that induces a regression test failure, we consider the following complexity features, which are defined based on the call graph of this test at the level of methods.

- Maximum depth of the call graph (abbreviated as $MaxD$). $MaxD$ represents the length of the longest chain of nested method calls of the JUnit test. Intuitively, the larger $MaxD$ is, the more complex the interaction between the test and the software under test is, and thus the more likely the failure is due to a bug in the product code.
- Number of methods called in the graph (abbreviated as $MethodNum$), which is the total number of methods called directly or indirectly by the JUnit test by counting each method only once. Intuitively, the larger $MethodNum$ is, the more likely the failure is due to a bug in the product code.

Change Feature. The change feature is concerned with the change between the current version and its previous version of the software under test. Intuitively, if the software under test undergoes a substantial revision, it is more likely that developers want to change some behavior of the software and thus a regression test failure is more likely due to the obsoleteness of the tests; but if the software under test undergoes a very light revision, it is more likely that developers do not want to change the behavior of the software and thus a regression test failure is more likely due to the product code. To consider this factor, we use the following change feature in our approach.

- File change (abbreviated as $FileChange$), which denotes the ratio of modification on the files that contain the methods called directly or indirectly by the failure inducing test. For a given test t, we use set $F(t)$ to represent the set of files that contain the methods called directly or indirectly by t. For a file f belonging to $F(t)$, we denote its previous version before the revision as f_b and its latter modified version as f_l. Furthermore, we use $|f_b|$ and $|f_l|$ to represent the number of lines of executable code (i.e., without counting lines containing only comments and/or blanks) of f_b and that of f_l, respectively. We then use $Change(f_b, f_l)$ to denote the number of different lines of executable code between f_b and f_l. Specifically, many tools (e.g., the unix command "Diff") can be used to compare two files and generate the different lines. Finally, we calculate $FileChange$ using the equation in Formula 1. Intuitively, the more total changes are involved in the call graph of a failure inducing test, the more likely the failure is due to an obsolete test.

$$FlieChange(t) = \frac{\sum_{\forall f \in F(t)} Change(f_b, f_l)}{\sum_{\forall f \in F(t)} maximum\{|f_b|, |f_l|\}} \quad (1)$$

Testing Features. The testing features are concerned with the testing results of all the executed tests. By using these features, our approach is able to consider the testing results of the whole test suite. In particular, we consider the following four testing features.

- Type of failure (abbreviated as $FailureType$), which denotes the type of the failing testing results returned by JUnit. In particular, $FailureType$ can be $Failure$, $Compile\_Error$ or $Runtime\_Error$, where $Failure$ denotes an assertion that is broken, $Compile\_Error$ denotes compiling problem when the compiler fails to compile the source code (including the product code and the test code) and $Runtim\_Error$ denotes a runtime exception when executing the product code with the test code.
- Count of plausible nodes in the call graph (abbreviated as $ErrorNodeNum$), which denotes the number of the methods that are called by the given failure inducing test and by at least another failure inducing test. Intuitively, if the call graph of the failure inducing test contains many such methods, the cause of the failure is more likely to lie in the product code than in the test code because one obsolete test may be unlikely to cause other tests to fail.
- Existence of highly fault-prone node in the call graph (abbreviated as $FaultProneNode$), which denotes whether the given failure inducing test calls a highly fault-prone method. Specifically, we define the highly fault-prone method as the methods that are called by more than half of the failed tests. As the highly fault-prone method is likely to contain bugs, the failure of a test calling such a method is more likely due to this method. That is, if the call graph of the failure inducing test contains such highly fault-prone methods, the cause of the failure is more likely to lie in the product code than in the test code.
- Product innocence (abbreviated as $ProInn$), which aims to measure the ratio of innocent product code involved in the call graph of the failure inducing test. For a given failure inducing test t, we use $M(t)$ to denote the set of methods called by t. Moreover, for a method m, we use $num_p(m)$ to denote the number of passed tests that call m and $num(m)$ to denote the total number of tests that call m. Then for any failure inducing test t, we calculate $ProInn$ using the equation in Formula 2, where k is the smoothing coefficient and set to be 1 in our approach. If a failure inducing test t has larger $ProInn$, most of the product code t calls is innocent (according to Formula 2), and thus the cause of the failure is more likely to be that t itself is obsolete. That is, the larger $ProInn$ is for a failure inducing test, the more likely the failure is caused by an obsolete test in the test code.

$$ProInn(t) = \prod_{\forall m \in M(t)} \frac{num_p(m) + 1}{num(m) + k} \tag{2}$$

Note that, although the collection of data for the four testing features requires executing the test suite, there is no need to instrument the product code to record coverage information. Based on the testing results returned by the JUnit testing framework, the data collection can be done through statically analyzing the call graph.

4.3 Failure-Cause Classification

We use the following steps to learn the classifier for failure-inducing tests. First, we collect the failure-inducing tests from the software repository. Second, we extract the values of the identified seven features and label each failure-inducing test. Finally, we train a classifier using the failure-inducing tests whose feature values and failure causes are known.

Collecting Failure-Inducing Tests. When a failure is observed in regression testing, developers can record the failing tests. Such information can be stored in the software repository as a part of artifacts during software development. From the repository, our approach collects these failure-inducing tests, which are the training instances used to train a failure-cause classifier in this paper.

Determining Feature Values and Failure Causes. For the seven identified features, we determine their values by analyzing the product code under test and the tests.

For the complexity features (i.e., $MaxD$ and $MethodNum$), we calculate their values using the call graph of the failure inducing test. Specifically, we use our previous work [38] and its corresponding tool Jtop [67] to implement the static call graph used in this paper.

To determine the value of $FileChange$, we first find the methods called by the failure inducing test, and then use the method signature to identify the corresponding source code in the previous version and that in the current version. After matching the two versions, we can calculate the value of $FileChange$.

The value of $FailureType$ can be directly obtained from the JUnit testing framework. To calculate the values of $ErrorNodeNum$, $FaultProneNode$, and $ProInn$, we first obtain the testing results of all tests from the JUnit testing framework, then mark methods in the call graph of the failure inducing test as either plausible or innocent, and finally calculate the three values.

To determine the cause of each regression test failure, we require developers evolving the software to label whether a regression test failure in the training set is due to the product code or the obsolete test.

Training a Classifier. Based on the training set (which contains a set of training instances with their features and labels), we are able to train a classifier for our target problem. Specifically, we adopt a typical machine-learning algorithm, the Best-first Decision Tree Learning algorithm [47,48]. The Best-first Decision Tree Learning algorithm is based on a decision tree model, which expands the "best" nodes first rather than in depth-first order used by C4.5 algorithm [43]. The "best" node is the one whose split will lead to maximum reduction impurity among all the nodes.

After training a classifier using the Best-first Decision Tree Learning algorithm, we use the classifier to classify the cause of future failure-inducing tests.

5 Experimental Studies

5.1 Research Questions

We have conducted three empirical studies to investigate the performance of the proposed approach in three scenarios.

In scenario (1), our approach constructs a classifier based on some regression test failures for a version of a program and uses the constructed classifier to classify the causes of other regression test failures for the same version. Thus, the first research question (**RQ1**) is as follows: Is our approach effective in classifying the causes of regression test failures when being applied within one version of each program?

In scenario (2), our approach constructs a classifier based on regression test failures for one version of a program and uses the constructed classifier to classify the causes of regression test failures for the subsequent version of the program. Thus, the second research question (**RQ2**) is as follows: Is our approach effective in classifying the causes of regression test failures when being applied between two versions of each program?

In scenario (3), our approach constructs a classifier based on one program and uses the constructed classifier to classify the causes of regression test failures for another program. Thus, the third research question (**RQ3**) is as follows: Is our approach effective in classifying the causes of regression test failures when being applied across different programs?

5.2 Studied Subjects

In our experimental studies, we used two non-trivial Java programs (i.e., Jfreechart and Freecol), whose product code and test code are available from SourceForge[5]. Jfreechart[6] is a Java application used to construct graphs and tables. Freecol[7] is a software game. Each of these two programs has several available versions whose test code is written in the JUnit framework. In our experimental studies, we used only the versions whose release dates are not in the same year so that the changes between versions are nontrivial.

Table 1 depicts the statistics of these two programs. Specifically, the first four columns depict the number of files, the total number of lines of executable code (by removing comments and blanks), the number of classes, and the number of methods in the product code, whereas the latter four columns depict the number of files, the total number of lines of executable code, the number of classes, and the number of methods in the test code.

As our approach is proposed to classify the cause of a regression test failure as a bug in the product code or an obsolete test, it is necessary to collect these two types of failures from practice.

[5] `http://sourceforge.net`
[6] `http://www.jfree.org/jfreechart/`
[7] `http://www.sourceforge.net/projects/freecol`

Table 1. Subjects in Our Studies

Program	Product Code				Test Code			
	#Files	#LOC	#Classes	#Methods	#Files	#LOC	#Classes	#Methods
Jfreechart 1.0.0	463	68,761	465	6,028	273	26,847	273	1,751
Jfreechart 1.0.7	538	80,927	540	7,335	356	42,052	356	2,634
Jfreechart 1.0.13	585	91,101	587	8,296	383	47,930	383	3,078
Freecol 0.10.3	578	94,031	579	6,757	85	13,022	85	493
Freecol 0.10.5	602	95,404	603	7,061	87	13,226	87	497

Developers usually release a version of the software when its tests cannot reveal any major faults in this version. Therefore, it is difficult to obtain faults that cause the regression test fail. Thus we manually injected faults in the product code by following some standard procedures [20]. Specifically, we randomly selected statements scattered in different files of the product code for each program and then generated faults by using mutant operators including negating a decision in conditional statements like "if" and "while", changing the values of some constants, and so on. After applying the test suite to the faulty product code, more failures would appear. We viewed the failures caused by the injected faults as due to bugs in the product code.

It is accessible to collect obsolete tests in practice. For either Jfreechart or Freecol, developers have released several versions during its development (as shown by Table 1). In regression testing, the tests for the old version may become obsolete for the new version and thus developers may need to modify existing tests or add new tests. Therefore, to access the obsolete tests in practice, we applied the test suite for a previous version of each program to the product code of the current version of the program. As the version of the used test suite is not consistent with the version of the product code, some tests may fail due to test obsoleteness. Here, we also refer to obsolete tests as faults in the test code, since they represent defects of the used test suite.

We summarize the statistical information of the faults in both product code and test code in Table 2, in which the first column depicts the abbreviation of the program, the following two columns depict the product code and the test code, and the latter three columns depict the number of tests in each test suite, the number of faults in the test code, and the number of faults in the product code.

5.3 Experimental Design

After preparing faulty product code and faulty test code, we collected the values of features of each program by analyzing the call graphs constructed by Jtop[8] and the testing results returned by JUnit. Based on the above data, we performed the following three studies.

In our first study, we constructed a classifier and evaluated the effectiveness of the constructed classifier by using the instances from the same version of a

[8] Jtop is a test management tool built in our previous research and is accessible at `http://jtop.sourceforge.net/`

Table 2. Faults in Our Studies

Abbreviation	Program	Test Suite	#Tests	# Faults in Test	# Faults in Product
J1	Jfreechart 1.0.7	Jfreechart 1.0.0	1,037	9	17
J2	Jfreechart 1.0.13	Jfreechart 1.0.7	1,706	8	18
F	Freecol 0.10.5	Freecol 0.10.3	362	82	74

program (i.e., Jfreechart 1.0.7, Jfreechart 1.0.13, or Freecol 0.10.5). For each version of a program, we used all its collected regression test failures, which are labeled by their causes (i.e., faults in the product code or faults in the test code), as instances and randomly split all the instances into a training set and a testing set. To reduce the influence of random selection, we used the 10 fold cross-validation technique to implement the selection process. Specifically, for a given set of data whose number of instances (i.e., regression test failures) is M, the 10 fold cross-validation technique divides the given set into 10 subsets of data equally so that each subset has $\frac{M}{10}$ instances. Then the 10 fold cross-validation technique uses each of these subsets as a testing set and the other subsets as a training set. In other words, the 10 fold cross-validation technique randomly selects $\frac{M*9}{10}$ instances of the given set as a training set, and takes the rest instances as a testing set. Moreover, the preceding process has been repeated 10 times within the 10 fold cross-validation technique. Based on each training set, our approach generates a classifier, which is evaluated by the instances of the testing set. Specifically, the Best-first Decision Tree Learning algorithm used in our approach is implemented on Weka 3.6.6[9], which is a popular environment for knowledge analysis based on machine learning and supports most machine-learning algorithms.

In our second study, we constructed a classifier by using the instances from a version of a program (i.e., Jfreechart 1.0.7) and evaluated the effectiveness of the constructed classifier by using the instances from the subsequent version of the program (i.e., Jfreechart 1.0.13). Specifically, we used all the regression test failures of Jfreechart 1.0.7 as the training set and all the regression test failures of Jfreechart 1.0.13 as the testing set. Based on the training set, our approach generates a classifier, which is evaluated by the instances of the testing set.

In our third study, we constructed a classifier by using the instances from one program (i.e., both the two versions of Jfreechart (including 1.0.7 and 1.0.13) or Freecol 0.10.5) and evaluated the effectiveness of the constructed classifier by using the instances from another program (i.e., Freecol 0.10.5 or both versions of Jfreechart). That is, we used all the regression test failures of one program as a training set and all the regression test failures of another program as a testing set. Based on each training set, our approach generates a classifier, which is evaluated by the instances of the testing set.

For each failed test in the testing set, we recorded (1) the number of failures correctly classified as faults in the product code, and (2) the number of failures

[9] `http://www.cs.waikato.ac.nz/ml/weka/`

correctly classified as faults in the test code. We then calculated the values of the metrics in Section 5.4 to evaluate our approach.

5.4 Evaluation Metrics

We used the following metrics to evaluate our approach.

- *OverAcc*, which denotes the overall accuracy of the proposed approach. This metric measures the likelihood of our approach to make a correct classification considering both causes of regression test failures.
- *AccFT*, which denotes the accuracy of classifying regression test failures due to faults in the test code. This metric measures the likelihood of our approach to make a correct classification considering only regression test failures due to faults in the test code.
- *AccFP*, which denotes the accuracy of classifying regression test failures due to faults in the product code. This metric measures the likelihood of our approach to make a correct classification considering only regression test failures due to faults in the product code.

An ideal approach should achieve values close to 1 for the *OverAcc* metric. Furthermore, for an ideal approach, the values of *AccFT* and the values of *AccFP* should not differ very much. It is not complete and reliable to compare two classification approaches based on only *AccFT* or *AccFP*. For example, supposing that there are totally 50 obsolete tests in the test code and 50 faults in the product code, if we classify all of them to be obsolete tests in the test code, the *AccFT* is 100% but its *AccFP* is 0% and its *OverAcc* is 50%. This approach is obviously bad as a good approach should have a high *OverAcc* with balanced *AccFT* and *AccFP*.

5.5 Threats to Validity

The threat to construct validity comes from the tests whose failures are due to the product code. It is hard to collect bugs in the product code in software evolution since developers usually release a software product after fixing bugs exposed by tests. To reduce this threat, we manually inject faults in the product code following the standard procedure [20] in software testing and debugging. The standard procedure is similar to mutant generation. We did not use existing mutation tools (e.g., MuJava[10]) to generate mutants because such tools usually generate a very large number of mutants [63]. The threat to internal validity comes from our implementation. To reduce this threat, we reviewed all the code before conducting our experiments. The threats to external validity lie in the subjects used in the studies and the impact of machine learning. In this paper, we used five versions of two open source Java programs, which are not necessarily representative of other programs. As we considered three scenarios, the

[10] `http://cs.gmu.edu/~offutt/mujava/`

overall workload of evaluation for our research can be already similar to or even more than that for an existing piece of research on test repair, from which our research stems. To further reduce the threat from subjects, we need to evaluate our approach on larger programs in other language (e.g., C#, C++) with more failure-inducing tests. Another external threat comes from machine learning, as our approach constructs a classifier by applying some existing machine-learning algorithm to some subjects. Although machine learning does not have the power to identify the cause-effect chain to pinpoint the differentiator between the product code and the test code, it is a highly generalizable way to generate a solution to pinpoint the preceding differentiator. For example, if we construct a solution based on some observations, the solution can be applicable to only subjects for which the observations hold. However, as a machine-learning-based approach has the ability to summarize observations from existing data, we reply on machine learning instead of inventing a classifier manually. Note that there may not necessarily be just one same classifier for all different subjects.

6 Results and Analysis

In this section, we first present the results and analysis of the three studies in Section 6.1, Section 6.2, Section 6.3, then give a decision-tree sample in Section 6.4, and finally summarize the main findings of our experimental studies in Section 6.5.

6.1 Study I – Within the Same Version

Fig. 3 presents the results of our first study. Specifically, the top sub-figure depicts the overall accuracy of classification results (i.e., *OverAcc*), whereas the bottom sub-figures depict the accuracy of classification failures due to faults in the test code (i.e., *AccFT*) and the accuracy of classification failures due to faults in the product code (i.e., *AccFP*). For simplicity, we use J1, J2, and F to represent Jfreechart 1.0.7, Jfreechart 1.0.13, and Freecol 0.10.5. In study I, the training instances and testing instances are collected from the same version of a program (i.e., version 1.0.7 of Jfreechart using the test suite of version 1.0.0, version 1.0.13 of Jfreechart using the test suite of version 1.0.7, version 0.10.5 of Freecol using the test suite of version 0.10.3).

Concerning the comparison of the overall accuracy (*OverAcc*) of our approach with 50%, which can be regarded as a random classification or a blind guess of failure causes, all of our approach's *OverAcc* values for the programs are are around 80%, much larger than 50%. Furthermore, the values of *AccFT* and *AccFP* are usually close to the corresponding values of *OverAcc* except for Jfreechart 1.0.7. That is to say, the classifier of our approach constructed by using one version of a program can usually classify the causes of regression test failures of the same version quite accurately. Our approach is not very effective in classifying the faults in the test code of Jfreechart 1.0.7. We suspect the reason to be that Jfreechart 1.0.7 has a small number of tests (test suite of version 1.0.0)

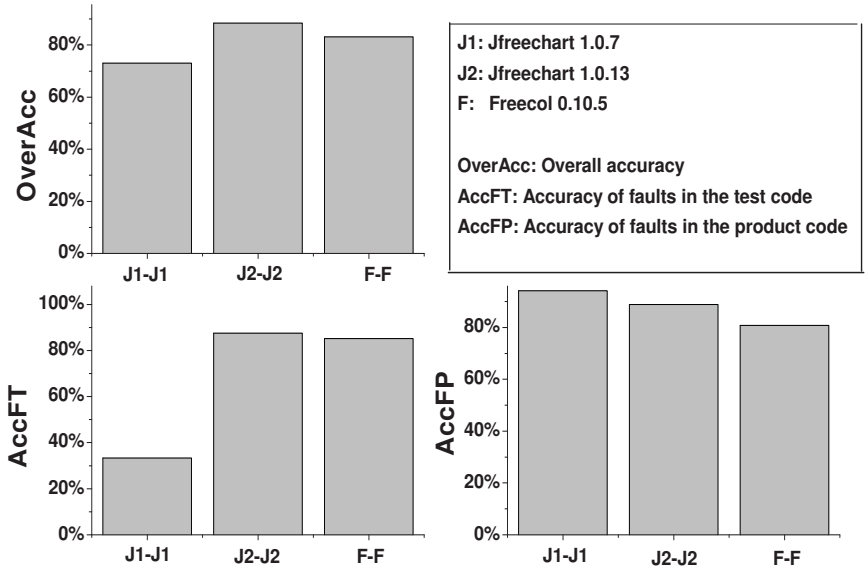


Fig. 3. Results of Study I

and a small number of faults in the test code so as to bias the classification results.

6.2 Study II – Between Versions

Fig. 4 depicts the results of our second study. The results for Jfreechart 1.0.7 are based on the training instances collected from version 1.0.7 of Jfreechart (using the test suite of version 1.0.0) and the testing instances collected from version 1.0.13 of Jfreechart (using the test suite of version 1.0.7).

The *OverAcc* value of our approach is higher than 50%, which is 96.15%. That is to say, the classifier constructed using some version of a program may be used to classify the cause of a regression test failure of another version of the program. Moreover, the results *AccFT* and *AccFP* for Jfreechart 1.0.7 are both close to 100%.

Furthermore, comparing the classification results of Jfreechart 1.0.13 (whose training instances and testing instances are all from Jfreechart 1.0.13) in Fig. 3 and those (whose training instances are from Jfreechart 1.0.7 but testing instances are from Jfreechart 1.0.13) in Fig. 4, the results (including *OverAcc*, *AccFT*, and *AccFP*) of the program increase. Although the differences between versions may harm the accuracy of a classifier, more instances are used to train the classifier in study II than study I, and thus our approach produces better results in study II than in study I.

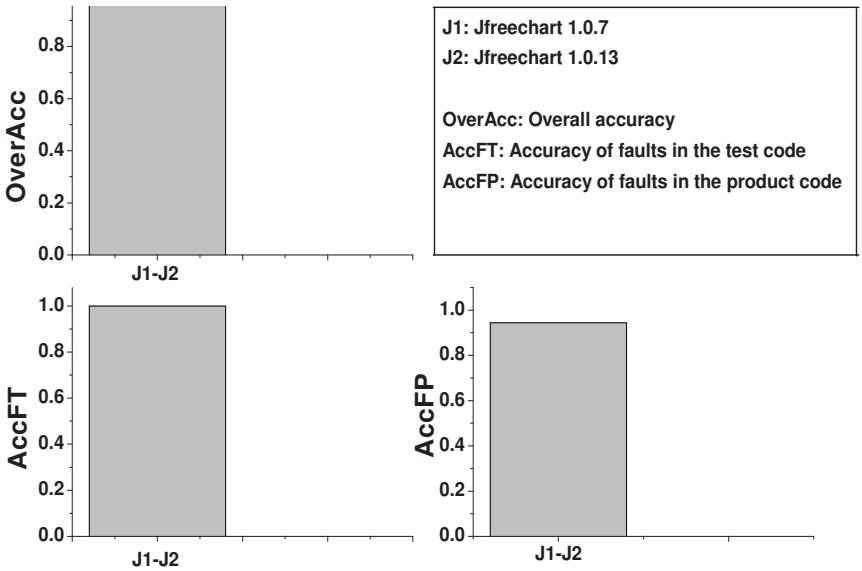


Fig. 4. Results of Study II

6.3 Study III – Across Programs

Fig. 5 depicts the results of our third study, where J is the abbreviation of Jfreechart (including its 1.0.7, and 1.0.13). The training instances are from version 1.0.7 and version 1.0.13 of Jfreechart and the testing instances are from version 0.10.5 of Freecol, or vice versa.

The *OverAcc* values of our approach are still higher than 50%, which are 68.18% and 71.15%, respectively. That is to say, our approach can still provide some help over the baseline. However, comparing to the classification results in Fig. 3 and Fig. 4, there are significant decreases. Furthermore, when closely examining the *AccFT* results and the *AccFP* results, the former may not be acceptable, because the values of *AccFT* are too low. In other words, the approach classifies too many failures as faults in the product code. We suspect the reason to be that the two programs used in the training set and in the testing set have significant differences in their structures so that the training process may have to face many noises.

The results of our third study are not satisfactory enough to be applied in practice, but it indicates the possibility that our approach may be applied between programs by improving the classifier using more features. It should be noted that cross-program validation is notoriously difficult for mining based approaches [53,72]. One possible way to alleviate this drawback is to set up a multi-program training set to prevent the trained classifier from being too specific to one program.

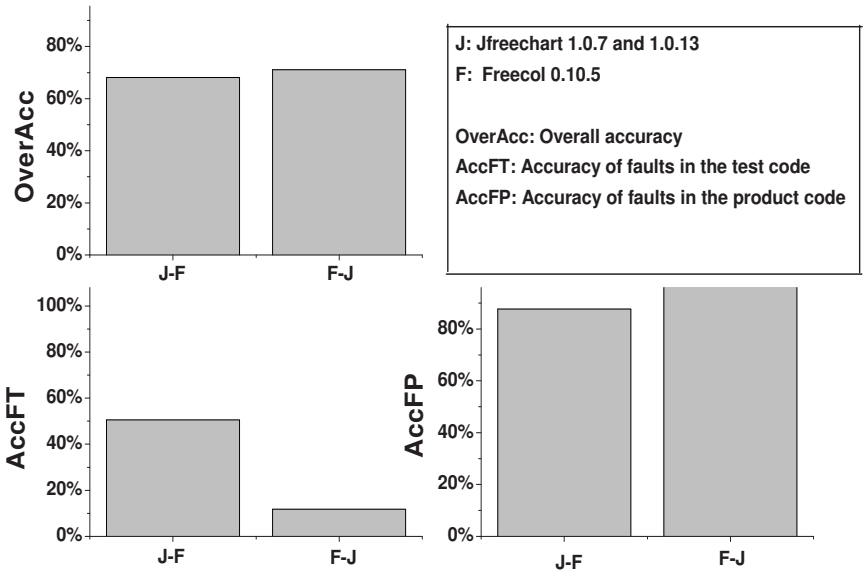


Fig. 5. Results of Study III

6.4 A Decision-Tree Sample

Fig. 6 presents a decision tree that are constructed using the all the regression test failures of Jfreechart 1.0.7 as training instances. In this decision tree, only three features (i.e., $FileChange$, $MaxD$, and $ErrorNodNum$) have contribution to differentiate the faults in the test code and the faults in the product code. In particular, if the values of $FileChange$ are smaller than 9.0, the causes of regression test failures are classified as faults in the product code; if the values of $FileChange$ are no smaller than 9.0 and the values of $MaxD$ are no smaller than 11.5, the causes of regression test failures are also classified as faults in the product code; if the values of $FileChange$ are no smaller than 9.0 and the values of $MaxD$ are smaller than 11.5 and the values of $ErrorNodeNum$ are smaller than 8.5, the causes of regression test failures are classified as faults in the test code. If the values of $FileChange$ are no smaller than 9.0 and the values of $MaxD$ are smaller than 11.5 and the values of $ErrorNodeNum$ are no smaller than 8.5, the causes of regression test failures are not clear and may be classified as faults in the product code or in the test code. We did not present the decision trees generated in the three studies due to the large number of generated decision trees.

6.5 Summary

In summary, the main findings of our experimental studies are as follows.

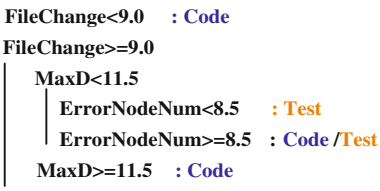


Fig. 6. Visualization of A Decision Tree

- First, our approach produces acceptable results when the training instances and the testing instances are from the same version of one program or from the different versions of one program.
- Second, when the training instances and the testing instances are from different programs, our approach is not as effective as being applied in the same program.

According to the two findings, our machine-learning based approach is generally effective to classify causes of regression test failures when the training instances and the testing instances are from the same program (including different versions and the same versions).

7 Discussion

In this section, we will discuss some issues that are related to our approach.

Adjustment of Features. Although our approach uses only seven features to construct a classifier, more features may be added to improve the effectiveness of the approach. Specifically, the complexity features and the change feature are collected by static analysis, whereas the testing features are collected by lightweight dynamic analysis. Dynamically collected features and statically collected features are complementary, and we will consider their cost and effectiveness in constructing a classifier in our future work. Furthermore, the seven features are intuitively correlated with the cause of a regression test failure, but our experimental studies did not investigate the impact of each feature on the classification results. Moreover, our intuition that how each feature impacts the classification results (e.g., a regression test failure is more likely due to buggy product code if the corresponding test has a small value of $FileChange$) is not encoded in the classifier since our intuition may not be consistent with the actual practice. To learn explicitly the contribution of each feature on classification results, we will conduct more experiments on various combination of these features in the future.

Applicability of Fault Localization Techniques. Our preliminary experimental results show that existing fault localization approaches can hardly be directly used to solve the problem of this paper. Conceptually, fault localization,

Table 3. Results of Tarantula in Classifying Test Failures

Program –Test Suite	Faults in the Test Code				Faults in the Product Code			
	#Total	#Correct	#Wrong	#Miss	#Total	#Correct	#Wrong	#Miss
Jfreechart 1.0.7 – Jfreechart 1.0.0	9	4	2	3	17	0	2	15
Jfreechart 1.0.13 – Jfreechart 1.0.7	8	1	7	0	16	15	0	1
Freecol 0.10.5 – Freecol 0.10.3	82	0	14	68	74	0	1	73

especially spectrum-based fault localization approaches may be extended to solve the problem in this paper. Therefore, we conducted a preliminary experimental study on two programs (with the same seeded faults) and their test suites in Table 2 using Tarantula [27], which is an effective and widely used spectrum-based fault localization approach. For each program and its test suite, Tarantula generates a ranked list of suspicious methods based on the descendent order of their suspiciousness, which measures the possibility that the methods contain faults. Formally, the ranked list of suspicious methods is denoted as $m_1, m_2, \ldots, m_n$, where m_i and m_j $(1 \leq i, j \leq n)$ denotes any methods of the program and its test suite. Supposed that $sus(m_i)$ and $sus(m_j)$ represent the suspiciousness of methods m_i and m_j, respectively, then $sus(m_i) \leq sus(m_j)$ iff $j \leq i$. If m_1 is a method in the product code and any method m_k in the test code satisfying that $sus(m_1) > sus(m_k)$, we deem that the faults are in the product code; if m_1 is a method in the test code and any method m_k in the product code satisfying that $sus(m_1) > sus(m_k)$, we deem that the faults are in the test code; otherwise, it is not clear where the faults are since both the method in the product code and the method in the test code have the largest suspiciousness. Based on this assumption, the classification results of Tarantula can be summarized as Table 3, where #Total shows the total number of faults in either the test code or the product code, #Correct shows how many faults (in either the test code or the product code) have been correctly classified by Tarantula, #Wrong shows how many faults have been incorrectly classified by Tarantula (i.e., the faults in the test code have been classified as faults in the product code, or the faults in the product code have been classified as faults in the test code), and #Miss shows how many faults (in either the test code or the product code) cannot be clearly[11] classified by Tarantula. According to this table, Tarantula can hardly precisely classify the cause of a failed test in most cases, especially when the faults are due to an obsolete test. Therefore, existing fault localization approaches cannot be directly applied to solve the problem in this paper. In the future, we will consider how to utilize the results of fault localization approaches to improve the classification.

Impact of Structure Changes. To collect the change feature, our approach is required to identify changes between versions, but some structural changes (e.g.,

[11] Sometimes methods in the product code and methods in the test code are assigned with the same suspiciousness and thus Tarantula cannot tell whether the faults are due to the test code or the product code.

renaming) may cause noise in matching the entities between versions. Changes on the program's behavior should be manifested on its test case, whereas changes only on the program's structure may not. Therefore, it is important to precisely map the entities between versions so as to reduce the noise in gathering the change feature. Refactoring [11,49] is an important type of changes in object-oriented software, which changes the structure of a program without affecting its behavior [13]. As there exist many refactoring tools [10,55], we will use these tools to match entities between versions in collecting the change feature so as to reduce the noise resulting from some structure changes in the future.

Other Machine-Learning Algorithms. In this paper we present and evaluate our approach using a typical machine-learning algorithm (i.e., Best-first Decision Tree algorithm), but there exist many machine-learning algorithms in the literature. Besides Best-first Decision Tree algorithm, we have implemented our approach using another algorithm the Naive Bayes algorithm [39], and found that the latter was much worse than the former. Besides machine-learning algorithms, there are many other factors (e.g., size of samples) that may influence the effectiveness of the proposed approach. As this paper is only a first step on classifying the cause of a regress test failure, we will further investigate the impact of these factors in our future work.

8 Conclusions and Future Work

In software evolution, when we apply an existing test suite to the modified software, some regression tests may fail. The failures of these regression tests may be due to buggy product code or obsolete test code. Before applying existing debugging techniques in the product code or test-repair techniques, it is necessary to determine whether a failure is due to the bug in the product code or obsolete tests. In this paper, we propose a machine-learning based approach, which collects values of seven features that may be related to failures of regression tests and then constructs a classifier by using a machine-learning algorithm (i.e., the Best-first Tree Learning algorithm). Furthermore, we evaluated this approach in three scenarios and found that the overall accuracy of our approach on correctly classifying regression failures is mostly about 80% when being applied within a program.

In future, we will identify more failure-related features to further improve the classification accuracy. We will also evaluate the proposed approach on a variety of projects written in different programming languages. As the empirical study did not evaluate the effectiveness of the given features on classification, we will evaluate which features play a leading role in the classification in our future work. Furthermore, we will work on the feasibility of establishing a discriminative model, aiming at classifying the causes of regression test failures based on features.

Acknowledgement. This research is sponsored by the National 973 Program of China No. 2009CB320703, the National 863 Program of China No. 2012AA011202, the Science Fund for Creative Research Groups of China No. 61121063, and the National Natural Science Foundation of China under Grant Nos. 91118004, 61225007, 61228203, and 61272157.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques, pp. 89–98 (2007)
2. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit state model checking. IEEE Transactions on Software Engineering 34(4), 474–494 (2010)
3. Bowring, J.F., Rehg, J.M., Harrold, M.J.: Active learning for automatic classification of software behavior. In: ISSTA, pp. 195–205 (2004)
4. Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: ICSE, pp. 480–490 (2004)
5. Black, J., Melachrinoudis, E., Kaeli, D.: Bi-criteria models for all-uses test suite reduction. In: ICSE, pp. 106–115 (2004)
6. Daniel, B., Dig, D., Gvero, T., Jagannath, V., Jiaa, J., Mitchell, D., Nogiec, J., Tan, S.H., Marinov, D.: ReAssert: a tool for repairing broken unit tests. In: ICSE, pp. 1010–1012 (2011)
7. Daniel, B., Gvero, T., Marinov, D.: On test repair using symbolic execution. In: ISSTA, pp. 207–218 (2010)
8. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: Reassert: suggesting repairs for broken unit tests. In: ASE, pp. 433–444 (2009)
9. Dean, B.C., Pressly, W.B., Malloy, B.A., Whitley, A.A.: A linear programming approach for automated localization of multiple faults. In: ASE, pp. 640–644 (2009)
10. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 404–428. Springer, Heidelberg (2006)
11. Dig, D., Marrero, J., Ernst, M.D.: Refactoring sequential Java code for concurrency via concurrent libraries. In: ICSE, pp. 397–407 (2009)
12. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Transactions on Software Engineering 28(2), 159–182 (2002)
13. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
14. Francis, P., Leon, D., Minch, M., Podgurski, A.: Tree-based methods for classifying software failures. In: ISSRE, pp. 451–462 (2004)
15. Galli, M., Lanza, M., Nierstrasz, O., Wuyts, R.: Ordering broken unit tests for focused debugging. In: ICSM, pp. 114–123 (2004)
16. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR (2002)
17. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering 38(1), 54–72 (2012)

18. Hao, D., Wu, X., Zhang, L.: An empirical study of execution-data classification based on machine learning. In: SEKE, pp. 283–288 (2012)
19. Hao, D., Xie, T., Zhang, L., Wang, X., Mei, H., Sun, J.: Test input reduction for result inspection to facilitate fault localization. Automated Software Engineering 17(1), 5–31 (2010)
20. Hao, D., Zhang, L., Pan, Y., Mei, H., Sun, J.: On similarity-awareness in testing-based fault-localization. Automated Software Engineering 15(2), 207–249 (2008)
21. Hao, D., Zhang, L., Wu, X., Mei, H., Rothermel, G.: On-demand test suite reduction. In: ICSE, pp. 738–748 (2012)
22. Hao, D., Zhang, L., Xie, T., Mei, H., Sun, J.: Interactive fault localization using test information. Journal of Computer Science and Technology 24(5), 962–974 (2009)
23. Haran, M., Karr, A., Orso, A., Porter, A., Sanil, A.: Applying classification techniques to remotely-collected program execution data. In: FSE, pp. 146–155 (2005)
24. Harman, M., Alshahwan, N.: Automated session data repair for web application regression testing. In: ICST, pp. 298–307 (2008)
25. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2(3), 270–285 (1993)
26. Høst, E.W., Østvold, B.M.: Debugging method name. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 294–317. Springer, Heidelberg (2009)
27. Jones, J.A., Harrold, M.J.: Empirical evaluation of tarantula automatic fault-localization technique. In: ASE, pp. 273–282 (2005)
28. Jose, M., Majumdar, R.: Cause clue clauses: Error localization using maximum satisfiability. In: PLDI, pp. 437–446 (2011)
29. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: ICSE (to appear, 2013)
30. Kim, S., Zhang, H., Wu, R., Gong, L.: Dealing with noise in defect prediction. In: ICSE, pp. 481–490 (2011)
31. Koskinen, J.: Software Maintenance Cost (2003), `http://users.jyu.fi/koskinen/smcosts.htm`
32. Lehman, M.M., Belady, L.A.: Program Evolution C Processes of Software Change. Academic Press, London (1985)
33. Leung, H.K.N., White, L.: Insights into regression testing. In: ICSM, pp. 60–69 (1989)
34. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: PLDI, pp. 141–154 (2003)
35. Mansour, N., El-Fakih, K.: Simulated annealing and genetic algorithms for optimal regression testing. Journal of Software Maintenance 11(1), 19–34 (1999)
36. Memon, A.M.: Automatically repairing event sequence-based gui test suites for regression testing. ACM Transactions on Software Engineering and Methodology 18(2), 1–35 (2008)
37. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering 32(11), 1–12 (2007)
38. Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., Rothermel, G.: A static approach to prioritizing Junit test cases. IEEE Transactions on Software Engineering 38(6), 1258–1275 (2012)
39. McCallum, A., Nigam, K.: A comparison of event models for naive bayes text classification. In: AAAI, pp. 41–48 (1998)
40. Mirzaaghaei, M., Pastore, F., Pezze, M.: Supporting test suite evolution through test case adaption. In: ICST, pp. 231–240 (2012)

41. Pinto, L.S., Sinha, S., Orso, A.: Understanding myths and realities of test-suite evolution. In: FSE, pp. 1–11 (2012)
42. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Automated support for classifying software failure reports. In: ICSE, pp. 465–474 (2003)
43. Quinlan, J.R.: Introduction of Decision Trees. Machine Learning (1986)
44. Rajan, A., Whalen, M.W., Heimdahl, M.P.E.: The effect of program and model structure on MC/DC test adequacy coverage. In: ICSE, pp. 161–170 (2008)
45. Robinson, B., Ernst, M.D., Perkins, J.H., Augustine, V., Li, N.: Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In: ASE, pp.23-32 (2011)
46. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices. Addison-Wesley (2003)
47. Shi, H.: Best-First Decision Tree Learning. Master Thesis, the University of Waikato (2007)
48. Stiglic, G., Kocbek, S., Kokol, P.: Comprehensibility of classifiers for future microarray analysis datasets. In: PRIB, pp. 1–11 (2009)
49. Taneja, K., Dig, D., Xie, T.: Automated detection of api refactorings in libraries. In: ASE, pp. 377–380 (2007)
50. Tillmann, N., Schulte, W.: Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. Microsoft Research. Technical Report (2005)
51. Wang, T., Roychoudhury, A.: Automated path generation for software fault localization. In: ASE, pp. 347–351 (2005)
52. Wang, X., Cheung, S.C., Chan, W.K., Zhang, Z.: Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In: ICSE, pp. 45–55 (2009)
53. Wang, X., Dang, Y., Zhang, L., Zhang, D., Lan, E., Mei, H.: Can I clone this piece of code here? In: ASE, pp. 170–179 (2012)
54. Wang, X., Zhang, L., Xie, T., Anvik, J.: Sun. J.: An approach to detecting duplicate bug reports using natural language and execution information. In: ICSE, pp. 461–470 (2008)
55. Weiβgerer, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE, pp. 231–240 (2006)
56. Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE, pp. 364–374 (2009)
57. Wloka, J., Ryder, B.G., Tip, F.: JUnitMx - A change-aware unit testing tool. In: ICSE, pp. 567–570 (2009)
58. Yang, G., Khurshid, S., Kim, M.: Specification-based test repair using a lightweight formal method. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 455–470. Springer, Heidelberg (2012)
59. Zaidman, A., Rompaey, B.V., Demeyer, S., Deursen, A.: Mining software repositories to study co-evolution of production & test code. In: ICST, pp. 433–444 (2008)
60. Zhang, H., Zhang, X., Gu, M.: Predicting defective software components from code complexity measures. In: PRDC, pp. 93–96 (2007)
61. Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H.: Bridging the gap between the total and the additional test-case prioritization strategies. In: ICSE, pp. 192–203 (2013)
62. Zhang, L., Hou, S., Guo, C., Xie, T., Mei, H.: Time-aware test-case prioritization using integer linear programming. In: ISSTA, pp. 213–223 (2009)

63. Zhang, L., Hou, S., Hu, J., Xie, T., Mei, H.: Is operator-based mutant selection superior to random mutant selection? In: ICSE, pp. 435–444 (2010)
64. Zhang, L., Kim, M., Khurshid, S.: Localizing failure-inducing program edits based on spectrum information. In: ICSM, pp. 23–32 (2011)
65. Zhang, L., Marinov, D., Zhang, L., Khurshid, S.: An empirical study of JUnit test-suite reduction. In: ISSRE, pp. 170–179 (2011)
66. Zhang, L., Marinov, D., Zhang, L., Khurshid, S.: Regression mutation testing. In: ISSTA, pp. 331–341 (2012)
67. Zhang, L., Zhou, J., Hao, D., Zhang, L., Mei, H.: Jtop: Managing JUnit test cases in absence of coverage information. In: ASE (Research Demo Track), pp. 673–675 (2009)
68. Zhang, L., Zhou, J., Hao, D., Zhang, L., Mei, H.: Prioritizing JUnit test cases in absence of coverage information. In: ICSM, pp. 19–28 (2009)
69. Zheng, X., Chen, M.H.: Maintaining multi-tier web applications. In: ICSM, pp. 355–364 (2007)
70. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and recommending API usage patterns. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 318–343. Springer, Heidelberg (2009)
71. Zhong, H., Zhang, L., Mei, H.: An experimental study of four typical test suite reduction techniques. Information and Software Technolgoy 50, 534–546 (2008)
72. Zimmermann, T., Nagappan, N., Gall, H.: Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: FSE, pp. 91–100 (2009)

Systematic Testing of Refactoring Engines on Real Software Projects

Milos Gligoric[1], Farnaz Behrang[2], Yilong Li[1], Jeffrey Overbey[2], Munawar Hafiz[2], and Darko Marinov[1]

[1] University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{gliga,yli147,marinov}@illinois.edu
[2] Auburn University, Auburn, AL 36849, USA
fzb0012@tigermail.auburn.edu, jeffreyoverbey@acm.org, munawar@auburn.edu

Abstract. Testing refactoring engines is a challenging problem that has gained recent attention in research. Several techniques were proposed to automate generation of programs used as test inputs and to help developers in inspecting test failures. However, these techniques can require substantial effort for writing test generators or finding unique bugs, and do not provide an estimate of how reliable refactoring engines are for refactoring tasks on real software projects.

This paper evaluates an end-to-end approach for testing refactoring engines and estimating their reliability by (1) systematically applying refactorings at a large number of places in well-known, open-source projects and collecting failures during refactoring or while trying to compile the refactored projects, (2) clustering failures into a small, manageable number of failure groups, and (3) inspecting failures to identify non-duplicate bugs. By using this approach on the Eclipse refactoring engines for Java and C, we already found and reported 77 new bugs for Java and 43 for C. Despite the seemingly large numbers of bugs, we found these refactoring engines to be relatively reliable, with only 1.4% of refactoring tasks failing for Java and 7.5% for C.

Keywords: Refactoring engines, Systematic testing, Test clustering.

1 Introduction

Refactorings [11] are behavior-preserving code transformations that developers traditionally apply to improve the design of existing code. Modern IDEs—such as Eclipse, NetBeans, or Visual Studio—contain refactoring engines that automate applications of refactorings. Previous studies [6, 8, 28, 45] show that most commonly applied refactorings include renaming program elements, extracting methods, and inlining methods. The list of refactorings is growing as researchers and practitioners recognize new patterns that are worth automating [5, 9, 10, 36, 48].

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 629–654, 2013.

Testing refactoring engines is an important yet challenging problem. It is important because bugs[1] in refactoring engines can affect programmer productivity, introduce errors in the code being refactored, and reduce the confidence of the programmer who may decide to perform manual refactorings that can be even more error-prone. It is challenging because refactoring engines require complex test inputs, i.e., programs/projects to be refactored; such test inputs are hard to generate using naive random generation or symbolic execution [14].

Automated testing of refactoring engines has gained attention in research [7, 12, 17, 35, 38, 39, 41]. Most proposed techniques require manually writing *test generators* that use sophisticated random or bounded-exhaustive generation to produce the required complex test inputs. Such techniques have had some impact on the research and practice of building refactoring engines, e.g., by finding real bugs in widely used IDEs [7, 12, 38, 39] or by affecting design of refactoring engines [15, 31, 34, 35]. However, such techniques also have several deficiencies. First, they require substantial manual effort for writing test generators. Second, the generated test inputs may not represent real refactoring scenarios; the generators often produce "corner cases" that IDE developers or users do not care about. Third, they do not provide any estimate of how reliable refactoring engines are for tasks on real software projects.

Instead of using artificially generated programs to evaluate refactoring engines, several research projects [4, 5, 37, 41, 44] use real programs. Spinellis [41] mentions testing the RENAME refactoring of his CScout refactoring engine on all identifiers in the Linux kernel. Independently, Thies and Steimann [44] tested two refactorings in Eclipse in a similar manner. However, these projects did not consider the overall process from applying refactorings to inspecting failures to reporting new, unique bugs, and they did not quantify the reliability of widely used refactoring engines such as those in Eclipse. While previous studies [7, 35, 38, 39] show (and our current study confirms) that systematic testing of refactoring engines can expose a large number of failures, it is important to map these failures to bug reports. Jagannath et al. [17] proposed a technique that clusters failures to help in inspection, but they evaluated the technique only on artificially generated programs. (Section 6 discusses related work in more detail.)

This paper makes two contributions.

End-to-End Approach: We propose testing refactoring engines and evaluating their reliability by combining techniques that systematically apply refactorings on a large number of places in real software projects [4, 5, 41, 44] and that effectively cluster the failures to a small number of (likely unique) bugs [17]. Our approach consists of the following steps: (1) given a set of projects, systematically apply refactorings in many places and collect failures where the refactoring

[1] The term "bug" used in this paper is more formally called a "fault", i.e., an error in the code of a refactoring engine, in contrast to a "failure", i.e., an error observed from an execution of the refactoring engine.

engine throws an exception or produces refactored code that does not compile[2], (2) split these failures into clusters such that all failures in the same/different cluster/clusters are likely due to the same/different underlying bug/bugs, (3) inspect randomly selected failures from the clusters, minimize them, identify non-duplicate bugs, and report them. We fully automated step 1, semi-automated step 2, and, for now, manually perform step 3.

While previous work explored some individual steps separately, our combined approach leads to a more effective, end-to-end methodology for evaluating refactoring engines. In contrast to techniques that automate test generation [7, 12, 17, 38, 39], our approach does not require manually writing test generators, finds bugs that occur in real refactoring tasks, and allows us to characterize reliability of refactoring engines for real refactoring tasks. We expect that bugs commonly found in real applications are more likely to be fixed than bugs discovered from artificially generated corner cases.

Evaluation: We use our approach to extensively evaluate the Eclipse refactoring engines for two programming languages—Java and C. Our study is the first to test *all* refactorings—23 for Java and 5 for C—currently implemented in Eclipse for these two languages. So far we have found 77 bugs in 21 refactorings for Java (not finding any bug in two refactorings) and 43 bugs in 5 refactorings for C, which is more bugs than any previous study that we are aware of [7, 12, 38, 39]. We reported these bugs to the Eclipse developers, who acknowledged our reports—"Thanks for opening all the useful bug reports. Much appreciated!" (`http://dev.eclipse.org/mhonarc/lists/jdt-ui-dev/msg01278.html`)—and have already fixed 8 of these bugs. Our clustering technique effectively reduces almost 15000 failures in Java and C to 356 clusters to be inspected. Moreover, we find refactoring engines to be relatively reliable, with the average rate of failing refactoring tasks being 1.4% for Java and 7.5% for C.

To the best of our knowledge, our study is the first to (1) evaluate this end-to-end approach of applying refactorings on real software projects and mapping the failures to unique bugs, (2) cluster failures of refactoring engines on real projects, (3) highlight the challenge of finding duplicate failures and bug reports, (4) show that this approach can be easily adopted for multiple programming languages unlike test generators that need to be written from scratch for each language, and (5) report failure rates for refactoring engines as a way to estimate their reliability. Our promising results provide motivation for the community to automate various steps from our approach, including minimization of programs that lead to failures [27, 33, 50] and searching for duplicate bug reports that involve programs as test inputs.

The key automated steps of our approach have been successfully evaluated by the ECOOP Artifact Evaluation Committee (`http://ecoop13-aec.cs.brown.edu/`) and found to meet expectations. Our main results with the links to the reported bugs are available online: `http://mir.cs.illinois.edu/rtr`.

[2] One can use other test oracles [7, 39] in addition to refactored code not compiling. Note that we check compilation only when the refactoring engine raises no warning that the refactoring should not proceed because some precondition is violated.

2 Example

As an example, we illustrate using our approach to test the Change Method Signature refactoring for Java. This refactoring takes as input one method and a set of changes to make to various parts of the method signature: visibility (e.g., `private`), return type, method name, and parameter list (add, remove, or reorder parameters). Changing the signature of one given method can lead to changing several other methods (e.g., those that override or are overridden by the given method) and can require changing the call sites to the method(s) being changed.

Our approach has three steps. In the first step, our automated tool compiles our corpus of real programs and finds *all* the program elements where a given refactoring can be applied (Section 3.1). For Change Method Signature, this corresponds to finding all the methods. For each such element, the tool then repeatedly applies the applicable refactoring tasks that pass the preconditions, records if there is a failure, and undoes the applied refactoring so that the next refactoring task can be applied. For Change Method Signature, our tool performs four refactoring tasks for each method: (1) changes visibility, (2) adds a parameter in first position, (3) removes the first parameter, and (4) reverses the order of parameters. (Section 3 has a detailed list.) Note that some tasks may not apply to some methods, e.g., a parameter cannot be removed if the target method has no parameters.

For the experiments, we use five popular Java projects: JPF, JUnit, log4j, Lucene, and Math (Section 5). On these projects, our tool performs a total of 28526 refactoring tasks for Change Method Signature. These tasks result in 565 failures. While the absolute number of failures is relatively large, the relative rate of failures is 565/28526=2.0%, i.e., only a relatively small fraction of all refactoring tasks result in a failure. Of these 565 failures, 555 are compiler errors denoting that the resulting program does not compile any more, and 10 are exception cases in which the refactoring engine throws an exception while applying the change. For each failure, our tool records where the refactoring is applied, the type of failure, and the messages produced by the failure—compiler error messages or exception stack traces.

It is worth pointing out that no prior study on testing refactoring engines [7, 12, 17, 35, 38, 39, 41, 44] report finding any exception case. At least one paper [7] explicitly states checking for such cases but finding no failure, and other papers use automated tools that would likely crash for uncaught exceptions and thus be observed by the researchers. Hence, the large number and diversity of real refactoring tasks, arising from applying our approach systematically on several open-source projects, enables us to discover these cases missed by previous work.

Even when the absolute number of failures is relatively large, many of them are due to the same underlying bug in the refactoring engine. Inspecting all the failures is prohibitively expensive and unnecessary to identify *unique* bugs. Since one of our goals is to identify new, unique bugs in the refactoring engine, we want to inspect a relatively small number of the failures that likely have different underlying bugs. A naive approach that randomly selects some number of failures to be inspected does not work well [17], and it is not obvious a priori

how many failures to randomly inspect. Thus, one can end up wasting time by inspecting several failures with the same underlying bug, or one can miss a bug by not inspecting any failure for that bug.

In the second step, our tool splits the failures into clusters based on the messages produced by the failure (Section 3.2). Ideally, clustering should satisfy two conditions: (1) all the failures in the same cluster should have the same underlying bug such that inspecting only one representative from each cluster will not miss any bug, and (2) the failures from different clusters should have different underlying bugs such that inspecting representatives from multiple clusters will not find duplicate bugs. To cluster the failures, we build on the idea of *abstract messages* [17]. This idea was previously proposed for automatically generated test inputs for refactoring engines but was not evaluated for failures on real refactoring tasks.

For Change Method Signature, our clustering splits 555 failures with compiler errors into 10 clusters (that have between 1 and 526 failures per cluster) and splits 10 failures with exceptions into 2 clusters (that have 4 and 6 failures). If one has insufficient resources to inspect all the clusters, one can prioritize the clusters based on the type of bugs one is looking for. For example, one can inspect clusters that have more failures before clusters that have fewer failures (thus looking for common bugs rather than looking for more "corner cases"), or inspect clusters that have failures arising from multiple projects before clusters that have failures arising from only one project (thus looking for bugs that are more common to be encountered by the users), or inspect clusters with exceptions before clusters with compiler errors (our anecdotal experience shows that Eclipse developers fix the exception cases faster as they may consider these bugs to be more severe).

In the third step, we manually analyze the clusters to identify and report non-duplicate bugs. For our running example, we inspect one randomly selected failure from each of these 10+2 clusters. This inspection involves two tasks: (1) minimizing the input project to understand the underlying bug and to prepare a bug report that makes debugging easier, and (2) identifying likely duplicates among the bugs in our clusters and bugs already in the Eclipse Bugzilla database. While there is research on automated minimization [27,33,50], we currently perform minimization manually. We experienced that minimizing a failure can sometimes take less time and effort than identifying duplicate bugs. Minimizing a failure took us 5–60 minutes, with an average around 10 minutes, while identifying duplicates sometimes took over 60 minutes, with an average around 15 minutes. For the examples in figures 1a and 1b, the minimization took 10 and 60 minutes, respectively. In the end, by inspecting 10+2 (compiler+exception) failures, we found 4+2 unique bugs, and of those 1+2 bugs were previously unreported in Eclipse Bugzilla.

We discuss in more detail the two bugs that lead to uncaught exceptions. Figure 1a shows the minimized code based on the Lucene project [24] that leads to a `NullPointerException` when the Change Method Signature refactoring is used to reorder the two parameters of the method `m`. In this case, the names

```
// C.java
class C {
  C(Object o) {}
  void m() {
    new C(new Object() {
      // Reorder parameters
      void m(int i, int j)
      {}
    });
  }
}
```

(a) NullPointerException

```
// .settings/org.eclipse.jdt.core.prefs
org.eclipse.jdt.core.compiler.source=1.4

// A.java
class A {
  // Remove parameter
  void add(int i) {
  }
}
```

(b) IndexOutOfBoundsException

Fig. 1. Examples of bugs found in CHANGE METHOD SIGNATURE by applying refactorings on Lucene and log4j projects, respectively

of the class, method, and parameters are not relevant for reproducing the exception. Figure 1b shows the minimized code from log4j [23] that leads to an `IndexOutOfBoundsException` when CHANGE METHOD SIGNATURE is used to remove the parameter of the method `add`; as opposed to the first bug, the method name must be `add` in order to be able to reproduce the bug. Additionally, the project must be using Java version 1.4 or lower (shown in the `settings` file in Figure 1b). Indeed, we find that reproducing some bugs requires more information about the project rather than just the source code of the program. This bug cannot be exposed by existing automated techniques for testing refactoring engines [7,12,38,39], because they focus on generating Java source code and not project configurations. In contrast, we found these bugs by applying refactorings on real projects.

While the minimized versions can look like "corner cases", the bugs actually arise on real code, and the IDE developers can use that information to prioritize fixing of the bugs. For example, `NullPointerException` related to Figure 1a arises in four refactoring tasks, whereas `IndexOutOfBoundsException` related to Figure 1b arises in six refactoring tasks.

3 Approach

This section describes in more detail our end-to-end approach for testing refactoring engines. Our approach consists of three main steps: (1) *collecting failures* discovers all refactoring tasks, runs these tasks, and outputs failing tasks (Section 3.1), (2) *clustering failures* splits failing tasks into clusters (Section 3.2), and (3) *inspecting failures* minimizes one failing task per cluster and finds duplicate failures to report new, unique bugs (since this step is currently manual, we do not discuss it in this section).

3.1 Collecting Failures

Figure 2 outlines the basic procedure for collecting failures. The procedure takes three inputs: the refactoring under test (RUT), a Java/C project containing the

```
1 collect_failures(refactoring, project, threshold):
2   elements = find_elements(refactoring, project)
3   for el in elements:
4     if is_reached(threshold): break
5     refactoring_tasks = create_refactoring_tasks(refactoring, project, el)
6     for task in refactoring_tasks:
7       configure_properties(task)
8       try:
9         if check_preconditions(task):
10          refactored_project = perform(task)
11        else: continue
12      except exc:
13        report("Failure: Refactoring threw an exception", exc, task)
14        continue
15      errors = compile(refactored_project)
16      if errors is not empty:
17        report("Failure: Refactored program failed check", errors, task)
18        continue
19      report("Success", task)
```

Fig. 2. Collecting failures for one given refactoring and project

program on which the refactoring will be applied, and a threshold that determines the maximum number of times to apply the RUT on the project. The procedure first finds the set of program *elements* in the given project on which the RUT can be applied and then computes a set of refactoring tasks for each element. For example, for Change Method Signature, the set of elements consists of all methods in the project, and the set of tasks can include changing method visibility, adding a parameter, removing a parameter, and reversing parameter order.

For each refactoring task, the procedure performs several steps in a loop. It first configures the properties for the refactoring task: in addition to the input project and program element, each refactoring can have a number of properties. For example, changing method visibility in Change Method Signature requires providing the new visibility: `private`, `protected`, `default`, or `public`. The specific property values depend on the particular refactoring task. For example, to actually change the method visibility we need to choose a new value for the visibility that differs from the old value, so different values can be provided for different refactoring tasks.

The procedure next checks if the refactoring task should proceed (line 9). In some cases the refactoring engine gives a warning that the refactoring could change the program behavior thus violating the definition that refactorings are behavior-preserving. For example, the refactoring engine could give a warning if we attempt to change visibility of a method to `private` when the method is called from outside its class. In those cases, our procedure does *not* proceed with the refactoring as checking the resulting program for compiler errors could produce many false positives because the problems do not arise from real bugs in the refactoring engine but from the ignored warnings. An alternative would be to proceed despite warnings but to check only whether the refactoring engine throws an exception and not whether the refactored program compiles.

```
1 # the procedure maintains a set called "abstractions"
2 # each abstraction maps a concrete message (exception or compiler error) to an abstract message
3 # applying abstractions to a concrete message returns either an abstract message or "cannot abstract"
4
5 cluster_failures(refactorings, projects, threshold):
6   # collect all failures
7   failures = {} # empty set
8   for rf in refactorings:
9     for pj in projects:
10      failures += collect_failures(rf, pj, threshold)
11
12  # cluster all failures
13  all_abstract_messages = {} # empty set
14  for failure in failures:
15    failure.abstract_messages = {} # empty set
16    for c_msg in failure.concrete_messages:
17      if apply(abstractions, c_msg) = "cannot abstract":
18        if can_automatically_abstract_messages(): # JDT tool
19          abstractions += automatically_abstract_message(c_msg, failure.task)
20        else: # current CDT tool
21          abstractions += ask_user_for_abstraction(c_msg)
22      a_msg = apply(abstractions, c_msg)
23      failure.abstract_messages += a_msg
24      all_abstract_messages += a_msg
25
26  clusters = {} # empty set of sets of failures
27  for rf in refactorings:
28    for type in { exception, compiler }:
29      for a_msg in all_abstract_messages:
30        cluster = { f ∈ failures | f.refactoring = rf ∧ f.type = type ∧
31                                   a_msg ∈ f.abstract_messages }
32        clusters += cluster
```

Fig. 3. Clustering failures for several given refactorings and projects

The procedure then performs the program transformation. Note that both this action and the previous action (lines 9 and 10) execute the actual RUT code from the refactoring engine. If these actions result in an uncaught exception, the procedure records a failure with an exception message.

If the refactoring task produced a refactored project, the procedure checks whether the new project compiles (line 15). One could optionally check other oracles [7,39], e.g., whether the refactored project still passes all its tests [41]. If there are any compiler errors, the procedure records a failure with all the error messages. Note that when one refactored project does not compile, there can be multiple compiler error messages, whereas when the refactoring engine throws an exception, there is only one message with a stack trace.

3.2 Clustering Failures

Figure 3 shows the procedure that runs a set of refactorings on a set of projects (up to the maximum number of refactoring tasks per refactoring and project pair), collects the failures, and then clusters the failures (lines 12 to 24). The goal of clustering is to reduce the number of failures that should be inspected to detect *unique* bugs.

The clustering first computes *abstract messages* from the concrete messages that were recorded with the failures. Each failure corresponds to a refactoring task that either threw an exception during the refactoring or produced compiler error(s) on the refactored project. The concrete messages are the actual strings, e.g., "`The type new MultivariateFunction(){} must implement the inherited abstract method MultivariateFunction.value(double[])`" and "`The type FieldValueHitQueue<T>.OneComparatorFieldValueHitQueue<T> must implement the inherited abstract method PriorityQueue<T>.lessThan(Object, T, T)`". The goal of abstracting these messages is to form clusters of failures that are likely due to the same underlying bug.

The procedure maintains a set of abstraction functions, each of which maps a concrete message (exception or compiler error) to an abstract message. For exceptions, our tool currently maps a failure to the top stack frame from the stack trace. For compiler errors, the abstractions are *regular expressions*. Our tool for Eclipse JDT automatically creates a regular expression from an object representing a compiler error during Eclipse execution; to obtain the error object, our tool reruns the refactoring task (line 19) and replaces all the arguments (e.g., `new MultivariateFunction(){}`) of the error message with "`.*`". For example, it can create a regular expression "`The type .* must implement the inherited abstract method .*`". Our tool for Eclipse CDT currently requires the user to manually provide a set of such regular expressions; we do not have full automation because some messages are more project specific because they are output from `make` not just compiler errors. These regular expressions typically ignore the project-specific details such as identifiers, file names, or line/column numbers. For each error, the tool checks whether the error matches one of the regular expressions; if not, the user is asked to provide a new expression. If yes, the regular expression itself is used as the abstract message. For example, the two messages from the previous paragraph are both abstracted to the same abstract message from this paragraph. Note that many regular expressions can be reused across refactorings. Across all failing refactoring tasks in our experiments, we had 112 automatically generated regular expressions for Java and 50 manually written regular expressions for C; it takes under a minute to manually write one regular expression, and we did not find it to be a big burden.

After the messages are abstracted, the clustering splits the failures into groups that have the same refactoring name (ignoring options for the refactoring task), the same type of failure (either exception or compiler error), and contain the same abstract message. As a result, one failure can belong to multiple clusters (known as "overlapping clustering" or "multi-view clustering" [16]), i.e., if a failure has multiple compiler errors, it is put in all the clusters that correspond to these errors. The expectation is that failures in the same cluster are likely due to the same bug, and failures in different clusters are likely due to different bugs. (Note that one failure by itself may be due to several bugs.) The clustering splits the failures based on the refactoring name because the chance is lower that failures for different refactorings are caused by the same bug, although the chance is not zero as some refactorings share code. Likewise, the clustering keeps

separate the clusters for exceptions and compiler errors because those clusters are unlikely to be caused by the same bug.

4 Implementation

This section describes how we implemented our approach for two refactoring engines in Eclipse—JDT [18] for Java and CDT [3] for C. While we implemented the approach only for Eclipse because of our familiarity with the infrastructure, the approach also applies to other IDEs.

4.1 Testing JDT

We implemented our tool as an Eclipse plug-in that supports testing all 23 refactorings available in the Eclipse refactoring menu (the first column of Figure 4)[3]. Our plug-in fully automates all the steps from figures 2 and 3 for the Eclipse JDT refactoring engine.

Our plug-in selects the set of relevant program elements for each refactoring based on the refactoring specification [32] (e.g., it selects methods for Change Method Signature). The second column of Figure 4 shows the precise set of elements that our tool selects by default. It selects only a subset of elements for some refactorings to match what was used in previous studies [7,12,38,39], e.g., for Rename these studies selected all fields, local variables, and methods but not types or packages. Our implementation offers a number of options that can select a superset or subset of the default set of elements, but our evaluation uses the default set.

Many refactorings have a number of properties that can be configured, e.g., for Convert Local Variable to Field the properties include: mark the field as final, mark the field as static, name for the field, location of initialization, and modifiers. By default our plug-in uses only one configuration of property values. Our experiments (Section 5) show that using one configuration suffices to find many new bugs in the current Eclipse refactoring engines; in the future, we plan to explore testing multiple configurations. The third column of Figure 4 lists the precise pairs `(property, value)` for all properties that our plug-in explicitly sets. For the properties that are not listed, our plug-in uses the default values that Eclipse provides. We select the values such that refactorings are likely to proceed and not raise warnings about violated preconditions, e.g., we rename a program elements to a name that is new to the project rather than some existing, conflicting name.

The main loop of our plug-in executes refactoring tasks and checks the results. These operations would be very slow if implemented naively by first creating a new Eclipse Java project for each refactoring task, then populating this project with the source code under test, refactoring the code, and compiling the entire refactored project to check for compiler errors. Our plug-in provides two important optimizations. First, it does not create a new Eclipse Java project for

[3] The order of the refactorings matches the order in the Eclipse refactoring menu.

Refactoring	Elements	(Property, Value)
Rename	fields$^{\mathcal{F}}$ local variables$^{\mathcal{L}}$ methods$^{\mathcal{M}}$	$^{\mathcal{F},\mathcal{L},\mathcal{M}}$new name, non-conflicting $^{\mathcal{L},\mathcal{M}}$update references, `true`
Move	instance methods$^{\mathcal{IM}}$ static methods$^{\mathcal{SM}}$	$^{\mathcal{IM},\mathcal{SM}}$delegate updating, `true` $^{\mathcal{IM},\mathcal{SM}}$deprecate delegates, `true` $^{\mathcal{IM}}$inline delegator, `true` $^{\mathcal{IM}}$use getter/setters, `true` $^{\mathcal{IM}}$target, non-primitive parameter types $^{\mathcal{SM}}$target, previous type in lexicographical order
Change Method Signature	methods$^{\mathcal{M}}$ parameters$^{\mathcal{P}}$	$^{\mathcal{M}}$visibility, `private` $^{\mathcal{P}}$default value for added, `null` $^{\mathcal{P}}$add/remove position, 0 $^{\mathcal{P}}$new order, reverse
Extract Method	expressions	new name, non-conflicting visibility, `public` replace duplicates, `true`
Extract Local	non-void expressions	declare final, `true` replace all occurrences, `true` new name, non-conflicting
Extract Constant	literals exp.s with literals method invocations	replace all occurrences, `true` visibility, `public` qualify references, `true`
Inline	constants$^{\mathcal{C}}$ local variables methods$^{\mathcal{M}}$	$^{\mathcal{C}}$remove declaration, `true` $^{\mathcal{M}}$delete source, `true`
Convert Local To Field	local variables	-
Convert Anonymous	anonymous classes	new name, non-conflicting declare static, `true`
Move Type To New File	non-local types	-
Extract Superclass	top level classes	create method stubs, `true` instanceof, `true` delete methods, `true` elements, all public methods
Extract Interface	types	annotations, `true` visibility, `public` replace, `true`
Use Supertype	types	destination, all supertypes
Push Down	types	element to push, a member
Pull Up	types	use keyword this, `true` override annotation, `true` destination, a supertype element to pull, a member
Extract Class	fields	create getter/setter, `true`
Introduce Param. Object	methods	top level, `false`
Introduce Indirection	methods	update references, `true`
Introduce Factory	methods	protect constructor, `true`
Introduce Parameter	expressions	-
Encapsulate Field	fields	visibility, `public` encapsulate declaring class, `true`
Generalize Declared Type	types	destination, a supertype
Infer Type Arguments	compilation units	-

Fig. 4. Default set of elements and (property, value) pairs for JDT

Refactoring	Elements	(Property, Value)
Rename	global variables local variables function parameters functions structure members macros	new name, non-conflicting
Extract Function	expressions single statements	new name, non-conflicting
Extract Local Variable	expressions	new name, non-conflicting
Extract Constant	literals	new name, non-conflicting
Toggle Function	functions	create header, `true`

Fig. 5. Default set of elements and (property, value) pairs for CDT

each task; instead, it creates one Eclipse Java project for the first task and then, after applying the refactoring and checking the results, it undoes the refactoring to restore the original project state. Undoing the refactoring is over an order of magnitude faster than creating a new Eclipse project. Currently, we rely on the Eclipse implementation of undo refactoring. However, as this implementation may be incorrect by itself, we could optionally copy the project to check the undo and to ensure that the project under refactoring is consistent before each refactoring task. Second, the plug-in does not compile the entire refactored project but only focuses on the file that contains the program element being refactored. Although this optimization significantly improves the performance, it may lead to false negatives as compiler errors may be in other affected files or the files that depend on the affected files. Our plug-in could be easily configured to compile the entire project after each refactoring task.

4.2 Testing CDT

Our implementation for Eclipse CDT, which targets the C and C++ programming languages, is similar to the implementation for JDT. For CDT we also implemented an Eclipse plug-in that supports all five C-specific refactorings available in CDT. We tested Eclipse 4.2.1 and CDT 8.1.1 (the Juno SR1 release of both Eclipse and CDT) [21].

Similar to our plug-in for testing the JDT refactorings, the set of relevant program elements for the CDT refactorings was derived from the refactoring specification. The default sets of elements and the default configurations for the refactorings are shown in Figure 5.

5 Evaluation

Our main goal was to evaluate how our proposed end-to-end approach helps in testing refactoring engines and estimating their reliability. This section describes

	Subject	Description	Version	LOC
Java	JPF [19]	Model checking tool	hg:960	95962
	JUnit [20]	Unit testing framework	git:r4.8.1-408-ge8b91fa	18199
	log4j [23]	Logging framework	svn:1406847	30058
	Lucene [24]	Text search engine library	3.5.0	129820
	Math [26]	Library of mathematics components	3.3.0	120424
C	GMP [13]	Arbitrary precision arithmetic library	4.3.2	81900
	libpng [22]	Official PNG reference library	1.2.6	33908
	zlib [51]	Lossless data-compression library	1.2.5	19855

Fig. 6. Subject programs used in the experiments

the projects that we used for testing Eclipse JDT and CDT refactoring engines, the failures that were collected, the clusters that were created, and the bugs that we reported to Eclipse Bugzilla.

5.1 Projects under Refactoring

Figure 6 shows the Java and C projects that we use in our evaluation. We tabulate the project name and the reference from which the project was obtained, a brief description of each project, version/revision number, and the number of (non-comment, non-blank) lines of code. We selected these projects because we were familiar with them, and they provide a diverse set of projects of various sizes (#LOC, #classes, #methods) and using different programming language features and design styles. For example, JUnit is a representative of a highly modular object-oriented design, whereas Math has a large number of local variables and constants.

5.2 Failure and Clustering Statistics

Figures 7 and 9 show the execution statistics from applying refactorings (in configurations from figures 4 and 5) on the selected set of projects for Java and C, respectively. For each refactoring and project, we tabulate the number of times that the refactoring is applied, the number of failures, and the total execution time (which includes finding the places where to apply the refactoring, applying the refactoring, and checking the refactored project).

JDT Results. We ran all JDT experiments on a 64-core Intel Xeon CPU L7555 @ 1.87GHz with 64GB of main memory, running Oracle Java version 1.7.0_04. In all runs we set the maximum number of refactoring tasks per file to 100 to limit the execution time. The runs still took over 200hrs of machine time overall.

The bottom of Figure 7 shows the ratio of the total number of failures and the total number of refactoring tasks applied on each project. The maximum ratio of 2.0% indicates that the Eclipse JDT refactoring engine is quite reliable, but there is still space for improvements.

Refactoring	JPF			JUnit			log4j			Lucene			Math		
	#Refact. Tasks	#Failures	Exec. Time [min]	#Refact. Tasks	#Failures	Exec. Time [min]	#Refact. Tasks	#Failures	Exec. Time [min]	#Refact. Tasks	#Failures	Exec. Time [min]	#Refact. Tasks	#Failures	Exec. Time [min]
Rename	20024	10	760	4025	8	69	5320	2	92	25897	24	626	26447	48	489
Move	1557	0	62	246	1	6	438	0	9	750	4	42	507	0	23
Change Method Signature	5021	289	429	5286	30	125	4261	65	129	6146	124	516	7812	57	440
Extract Method	26405	182	156	6183	28	27	8770	72	32	39468	152	206	39177	234	265
Extract Local	34834	0	245	4460	0	22	11020	0	46	50278	0	353	50796	0	393
Extract Constant	16752	0	183	2355	0	13	5444	0	26	26965	0	245	29997	0	257
Inline	18446	180	329	3745	60	41	4853	34	48	23499	569	323	21978	662	174
Convert Local To Field	7605	1	58	1041	0	4	2307	4	7	13802	8	58	2934	0	17
Convert Anonymous	146	1	4	143	0	1	7	0	0	293	38	3	314	5	5
Move Type To New File	198	19	10	440	5	9	82	3	1	550	22	24	165	2	5
Extract Superclass	248	0	74	419	0	11	87	0	2	232	0	24	590	0	43
Extract Interface	1251	237	69	730	31	15	332	43	7	1365	725	51	1122	121	29
Use Supertype	940	44	48	257	4	3	278	0	3	1195	24	31	814	13	19
Push Down	2666	79	353	387	11	9	529	7	8	2932	72	328	2001	14	51
Pull Up	3146	34	145	692	14	9	1774	2	11	5550	25	216	1657	87	40
Extract Class	61	2	1	363	45	2	92	15	1	221	22	4	2033	490	48
Introduce Param. Object	4590	114	1258	2421	105	73	2036	30	83	5185	233	515	7793	519	743
Introduce Indirection	4418	30	170	1813	4	37	1248	2	28	4835	41	200	4768	44	112
Introduce Factory	776	53	9	197	7	2	297	9	2	1069	62	13	1140	100	17
Introduce Parameter	895	359	12	2711	298	64	1711	110	42	2365	556	64	1721	194	44
Encapsulate Field	2749	37	50	563	3	6	977	71	11	4309	109	71	2885	10	41
Generalize Declared Type	2734	314	280	1285	157	15	2506	544	24	1240	144	37	2638	65	35
Infer Type Arguments	886	8	8	330	1	2	250	0	0	817	10	5	970	72	10
$\sum$	156348	1993	4713	40092	812	565	54619	1013	612	218963	2964	3955	210259	2737	3300
#Failures/#Refact. Tasks	1.3%			2.0%			1.9%			1.4%			1.3%		

Fig. 7. Execution statistics of our JDT plug-in on a set of Java projects

			Compiler Error					Exception				
Refactoring	#Refact. Tasks	#Failures	#Failures	#Clusters	Min Cluster	Max Cluster	#Bugs	#Failures	#Clusters	Min Cluster	Max Cluster	#Bugs
Rename	81713	92	90	5	1	54	‡6	2	1	2	2	1
Move	3498	5	5	2	1	4	1	0	0	-	-	0
Change Method Signature	28526	565	555	10	1	526	8	10	2	4	6	2
Extract Method	120003	668	665	14	1	390	5	3	1	3	3	0
Extract Local	151388	0	0	0	-	-	0	0	0	-	-	0
Extract Constant	81513	0	0	0	-	-	†1	0	0	-	-	0
Inline	72521	1505	1475	42	1	790	12	30	3	2	23	0
Convert Local To Field	27689	13	13	9	1	5	2	0	0	-	-	0
Convert Anonymous	903	44	29	8	1	10	2	15	1	15	15	0
Move Type To New File	1435	51	50	22	1	18	5	1	1	1	1	0
Extract Superclass	1576	0	0	0	-	-	0	0	0	-	-	0
Extract Interface	4800	1157	1143	16	1	725	4	14	1	14	14	0
Use Supertype	3484	85	85	21	1	16	6	0	0	-	-	0
Push Down	8515	183	183	11	1	121	6	0	0	-	-	0
Pull Up	12819	162	45	10	1	23	3	117	2	3	114	0
Extract Class	2770	574	574	16	1	275	3	0	0	-	-	0
Introduce Param. Object	22025	1001	839	15	1	455	2	162	4	2	140	0
Introduce Indirection	17082	121	72	7	4	31	1	49	3	4	32	2
Introduce Factory	3479	231	231	7	1	223	3	0	0	-	-	0
Introduce Parameter	9403	1517	0	0	-	-	0	1517	2	1	1516	‡3
Encapsulate Field	11483	230	212	10	1	94	8	18	1	18	18	0
Generalize Declared Type	10403	1224	1176	22	1	339	8	48	3	6	26	1
Infer Type Arguments	3253	91	7	7	1	3	1	84	2	83	84	2
$\sum$	680281	9519	7449	254			**87**	2070	27			**11**
#Failures/#Refact. Tasks	1.4%											

Fig. 8. Failure and cluster statistics for Java projects. (The number of bugs is likely higher; while we minimized one failure from each of 281 clusters, we have not checked duplicates for 141 minimized failures.) †The refactoring implements too strong precondition. ‡We reported two bugs that had the same stack trace but result in different compiler errors in the latest version of Eclipse.

Figure 8 shows additional statistics about failures. The column "#Refact. Tasks" shows the number of refactoring tasks performed across all the projects, and the column "#Failures" shows the total number of failures. The next two groups of columns split the results for the failures that have compiler errors or exceptions. Each group tabulates the number of failures, the number of clusters, the minimum and maximum sizes of clusters (measured by the number of failures), and the number of bugs we found based on these clusters.

We believe that the total of 680281 refactoring tasks cover a diverse spectrum of refactoring tasks and allow us to identify bugs that can be encountered in

Refactoring	GMP			libpng			zlib		
	#Refact. Tasks	#Failures	Exec. Time [min]	#Refact. Tasks	#Failures	Exec. Time [min]	#Refact. Tasks	#Failures	Exec. Time [min]
Rename	6688	555	739	2395	43	118	1569	0	9
Extract Method	16742	1176	579	5092	548	172	1496	58	24
Extract Local	5893	363	1240	2660	554	65	3473	406	59
Extract Constant	18788	387	902	2208	142	62	2800	331	28
Toggle Function	1434	403	10	302	293	2	332	167	1
$\sum$	49545	2884	3470	12657	1580	419	9670	962	121
#Failures/#Refact. Tasks	5.8%			12.4%			9.9%		

Fig. 9. Execution statistics of our CDT plug-in on a set of C projects

practice. The total number of failures is 9519, which may look large but is a relatively small fraction of the total number of refactoring tasks.

These failures are split into a total of 281 clusters: 254 compiler error clusters and 27 exception clusters. Clusters vary in size from 1 to 1516 failures, with the median and mean of 5 and 40.3, respectively. Recall that the cluster size can be used to prioritize inspection and/or fixing of bugs, and the same failure can appear in multiple clusters. For example, consider the two exception clusters for Infer Type Arguments. One cluster has all 84 failures, and the other cluster has 83 failures. It means that 83 failures have two messages each, and one failure has only one message (that abstracts to the same abstract message as one of the two messages from the other failures).

CDT Results. We ran all CDT experiments on an Intel Xeon Quad Core CPU X3440 @ 2.53GHz with 16GB of main memory.

For Rename, we run refactoring tasks on all C files in a given project. For each file, we attempt to rename at most 50 local variables, 50 function parameters, 20 global variables, 20 function names, and 20 macros. Across all three projects, the Rename refactorings run for a total of 866 minutes—overall, 10652 refactorings are attempted with 598 failures. Of these failures, 42% are compiler errors, while 58% are exception failures. We find a larger percentage of exceptions in CDT, presumably because it is less stable than JDT.

For Extract Function, we attempt to extract at most 100 statements and 100 expressions per C file. Out of 23330 attempts, 1782 fail, with 1453 compiler error failures and 329 exception failures. The total run takes 775 minutes. For Extract local Variable, Extract Constant, and Toggle Function, we attempt to extract at most 100 expressions, literals, and functions per C file, respectively. Across all refactorings, libpng has the highest failure rate with 12.4%, followed by zlib and GMP.

While we only check that the refactored program compiles, one can use other oracles [7, 39]. For example, for the Rename refactoring on GMP, we ran tests

			Compiler Error					Exception				
Refactoring	#Refact. Tasks	#Failures	#Failures	#Clusters	Min Cluster	Max Cluster	#Bugs	#Failures	#Clusters	Min Cluster	Max Cluster	#Bugs
Rename	10652	598	252	3	1	173	2	346	2	3	343	0
Extract Method	23330	1782	1453	34	1	435	21	329	8	1	56	3
Extract Local	12026	1323	754	11	3	412	7	569	4	6	263	3
Extract Constant	23796	860	142	3	1	84	2	718	3	125	426	2
Toggle Function	2068	863	9	2	1	8	1	854	5	23	409	2
$\sum$	71872	5426	2610	53			**33**	2816	22			**10**
#Failures/#Refact. Tasks	7.5%											

Fig. 10. Failure and cluster statistics for C projects

('`make test`') in addition to compilation ('`make`'). However, this did not produce any extra errors and was taking too much time, so we did not run tests for other cases. In the future, we plan to evaluate our approach with other oracles.

Figure 10 shows additional statistics about failures. We had 75 clusters in total: 53 compiler errors clusters and 22 exception clusters. These clusters vary in size from 1 to 435, with the median and mean of 27 and 79.5, respectively.

5.3 Bugs

After clustering all the failures, we inspected one, randomly selected failure from each of 281+75 clusters. We first minimized the project under refactoring such that the failure is preserved in the minimized version. We performed minimization manually, which took between a few minutes and 1hr, with the average of around 10min. In the future, we plan to evaluate the existing automated minimization techniques [27, 33, 50].

After we prepared a minimized version, we want to check whether it is a new, unique bug. We compared the minimized version with the other bugs that we found and also searched through the Eclipse Bugzilla database to ensure that the bug we found had not been reported before. This search for duplicates is also performed manually and took on average 15min per bug. (So far we have performed the search for 140 of 281 clusters for JDT and all 75 clusters for CDT.) Our goal was to report as few duplicates as possible, and we found it somewhat harder to search for duplicates than to minimize the project. One could consider searching for duplicates directly from failures, even before minimization, but our experience showed that the result is not obtained faster, e.g., searching based purely on compiler errors does not provide a good result. The existing techniques [1, 40, 47] for searching duplicate bug reports mostly use natural language processing and do not focus on searching for programs that are inputs to refactoring engines. We leave it as future work to explore automated

search for duplicates in this context. We point out that our study is the first to raise this concern; previous related studies [7, 12, 17, 35, 38, 39, 41, 44] did not report the effort for inspecting duplicates, presumably because (1) they found a smaller number of bugs than we found, (2) the number of bug reports for Eclipse was smaller at the time when they searched for duplicates than it was when we searched for duplicates, and/or (3) they did not search for duplicates.

So far we have reported a total of 77 bugs in JDT and 43 bugs in CDT. Each report includes the minimized example on which the bug can be reproduced. Our work is ongoing; we have 141 more Java minimized examples to check for duplicates and plan to run our tool for more projects in the future. The updated list of our reports is available online: `http://mir.cs.illinois.edu/rtr`.

Java Results. Figure 11 shows summary information about the bugs we have reported for Eclipse JDT so far. The first column lists the refactorings. The next column lists how many of the bugs we found are likely duplicates (either of previously reported bugs in Bugzilla or among our own clusters), which we did not report. The next group of columns lists how many reports we submitted and the current status of those reports in Bugzilla (`NEW` - the bug is reported but not yet considered by the Eclipse developers, `ASSIGNED` - the bug is confirmed and assigned to a developer, `FIXED` - the bug is fixed, and `DUPLICATE` - the bug is marked as a duplicate by the Eclipse developers).

The last row of the table summarizes the results: of the 77 bugs reported, 8 were already fixed, 62 assigned as real bugs, 4 marked as duplicates, and the rest were not inspected. We have noticed that the developers were more responsive if a reported bug causes an exception rather than a compiler error.

The remaining groups of columns for Java show the results for reproducing in two other IDEs, specifically NetBeans and IntelliJ IDEA, the bugs we found in Eclipse. The goal is to find how many of these bugs appear in one IDE but not another. Presumably the developers of an IDE may want to prioritize more the bugs that are unique to their IDE. We attempted to run on NetBeans and IntelliJ each minimized example that we used as part of our bug reports for Eclipse. We find that some of the bugs from Eclipse do not apply in other IDEs (e.g., because they do not have an equivalent refactoring or have too strong preconditions for the refactoring). Of the bugs that do apply, some are shared between independent implementations of refactorings, 24 between Eclipse and NetBeans, and 26 between Eclipse and IntelliJ (although not the same as NetBeans). A number of bugs (12) is shared even for all three IDEs. Of the bugs that could potentially apply, a number of bugs from Eclipse do not appear in the other IDEs. Note that this does not imply that the other IDEs are more reliable as we did not evaluate their bugs on Eclipse. Indeed, our goal was to find how bugs we found are shared among refactoring engines rather than to compare IDEs.

Anecdotal Experience. We found out that even duplicate reports can help developers, confirming some published results [1]. After inspecting a failure from one of the exception clusters, we discovered that the bug had been reported

	Java									C		
	Eclipse		Bugzilla				NetBeans	IntelliJ		Eclipse	VAX	XRef
Refactoring	#Likely Duplicate	#Reported	#NEW	#ASSIGNED	#FIXED	#DUPLICATE	Reproducible? yes/no/na	Reproducible? yes/no/na	Ecl.∩NetB.∩In.J	#NEW Eclipse	Reproducible? yes/no/na	Reproducible? yes/no/na
Rename	2	5	2	3	0	0	0/5/0	1/4/0	0	2	1/1/0	0/0/2
Move	0	1	0	1	0	0	1/0/0	0/1/0	0	-	-	-
Change Meth. Sig.	7	3	1	1	1	0	0/2/1	0/2/1	0	-	-	-
Extract Method	0	5	0	4	1	0	4/1/0	2/3/0	2	24	17/7/0	5/2/17
Extract Local	0	0	0	0	0	0	0/0/0	0/0/0	0	10	0/0/10	0/0/10
Extract Constant	0	1	0	1	0	0	0/0/1	0/1/0	0	4	0/0/4	0/0/4
Toggle Function	-	-	-	-	-	-	-	-	-	3	0/0/3	0/0/3
Inline	5	7	0	7	0	0	4/3/0	4/3/0	3	-	-	-
Loc. To Field	0	2	0	2	0	0	1/0/1	0/2/0	0	-	-	-
Anon. To Nested	0	2	0	2	0	0	1/1/0	0/2/0	0	-	-	-
Move Type	3	2	0	2	0	0	0/2/0	1/1/0	0	-	-	-
Extract Superclass	0	0	0	0	0	0	0/0/0	0/0/0	0	-	-	-
Extract Interface	0	4	0	4	0	0	0/1/3	3/1/0	0	-	-	-
Use Supertype	1	5	0	4	0	1	2/2/1	2/3/0	1	-	-	-
Push Down	2	4	0	3	0	1	3/1/0	1/3/0	1	-	-	-
Pull Up	0	3	0	2	1	0	2/1/0	2/1/0	2	-	-	-
Extract Class	0	3	0	3	0	0	0/0/3	0/0/3	0	-	-	-
Intro. Param. Obj.	0	2	0	2	0	0	0/2/0	1/1/0	0	-	-	-
Intro. Indirection	0	3	0	3	0	0	0/0/3	0/0/3	0	-	-	-
Intro. Factory	0	3	0	3	0	0	3/0/0	1/2/0	1	-	-	-
Intro. Param.	0	3	0	1	0	2	1/2/0	1/2/0	0	-	-	-
Encapsulate Field	1	7	0	5	2	0	2/5/0	5/2/0	2	-	-	-
Gen. Decl. Type	0	9	0	8	1	0	0/0/9	2/6/1	0	-	-	-
Infer Gen. Type	0	3	0	1	2	0	0/0/3	0/3/0	0	-	-	-
$\sum$	21	**77**	3	62	8	4	24/28/25	26/43/8	12	**43**	18/8/17	5/2/36

Fig. 11. Number of bugs for each refactoring from the Eclipse refactoring menu

previously. However, the original bug reporter explicitly stated being unable to create a small example that causes the exception. After we added our minimized example to the bug report, it was fixed within a day (by adding one line and updating one line), more than 4 years after the bug had been originally reported.

We also found out that some refactorings are quite reliable. As can be seen from Figure 11, we did not report any new bug for the Extract Local Variable and Extract Superclass refactorings. Suspecting that our configurations (Section 4.1) may be incorrect for these refactorings, we modified the configurations and rerun the refactorings but still did not find any failure. In the

future, we would like to explore many more additional configurations for these refactorings and to investigate the implementation of these refactorings to identify the reasons for their reliability.

C Results. Figure 11 (rightmost columns) shows summary information about the bugs we have reported for Eclipse CDT so far. The "Eclipse" column lists the number of bugs, and they are all still marked `NEW` in Bugzilla. We tried to manually reproduce the 26 Rename and Extract Function CDT bugs in two other refactoring engines: Visual Assist X (VAX) [46] and XRefactory [49]. The other three refactorings are not supported in these refactoring engines. VAX is a plugin that provides refactorings for Visual Studio; XRefactory is a plugin for Emacs, xEmacs, and jEdit. We used VAX running on Visual Studio 2008 and XRefactory version 2.0.14 running on Emacs version 24.1.

Of the two Rename bugs in CDT, one was about renaming functions in system/external libraries (e.g., `printf`), and the other one was about renaming a macro in a file that has been declared or used in another file. VAX successfully handled the first case, but failed in the second case. The bugs were not applicable to XRefactory, which did not make any changes on the given inputs. Interestingly, XRefactory does not allow renaming of any function except `main`. This is obviously a bug, but not exactly the one that we identified in CDT. Hence, we marked this case as not applicable.

Of the 24 bugs in Extract Function, 17 could be reproduced in VAX. Of these bugs, 8 produced the same outputs as CDT after extraction. The remaining 9 produced outputs that were different from CDT outputs, but they were incorrect too. 6 failures were related to incorrect return type of the functions. For example, when a user attempts to extract an assignment expression with VAX, the extracted function has a boolean return type, even if the assignment's type is not boolean. CDT also introduces incorrect return type for an Extract Function refactoring: it incorrectly determines the return type of a pointer variable to be a non-pointer variable of that type.

We could not apply 17 out of 24 Extract Function bugs in XRefactory. Most of them (16 out of 17) were about applying Extract Function on expressions, whereas XRefactory only allows extraction on statements. Another case was about extracting multiple configurations of the C preprocessor; in this case, XRefactory did nothing. Among the remaining 7 cases that were applicable, 5 were buggy. These cases failed while attempting to extract a statement related to a macro definition, a statement containing a variadic function, or a goto statement to a function.

We also looked at the quality of our clustering approach. In GMP, we found one duplicate compiler error bug across 6 different clusters and another duplicate compiler error bug across other 3 different clusters. We also found 4 different compiler error bugs within one cluster. Predictably, there are more bugs in the extract refactorings, because they are more complex. The search for duplicates is easier in CDT as it has fewer overall bugs reported for it. It is also less actively developed, so we did not see any change in Bugzilla for our CDT bug reports.

Application to OpenRefactory/C. We are actively integrating the approach described in this paper to test OpenRefactory/C [30], a new refactoring engine for C that we are developing. The approach is established as an essential part of developer testing: we are accumulating a set of well-known, open-source C projects, and refactorings are not eligible to be deemed "production quality" until they have successfully passed on those projects.

To use our approach for continuous developer testing, we have found it helpful to keep the number of tests relatively small (e.g., 50–100), at least during the early stages of testing. This typically produces just a few clusters, which the developer can investigate and fix immediately. Test results are persisted outside of Eclipse runs (currently in a database), which allows the developer to re-run failed tests after fixing a bug, or to continue running tests where a previous test run stopped. At the time of this writing, only one refactoring—Rename—in OpenRefactory/C is mature enough to be continously tested using this approach. Applying it to GMP, libpng, and zlib has already identified seven bugs: four bugs in the Rename refactoring itself (one unexpected exception) and three bugs in the supporting infrastructure (including one unexpected exception).

Use Frequency vs. Failure Rate. Several researchers have studied the frequency of refactoring use in Java IDEs [6, 8, 28, 45] and ranked Rename, Extract Local Variable, Inline, Extract Method, and Move as the top five most commonly performed automated refactorings in practice. It would be reasonable to expect the most commonly used refactorings to have fewer bugs and thus a smaller failure rate. However, our study does not find a very strong correlation between the frequency of use and failure rate of a refactoring. While the top five used refactorings are also among the most reliable refactorings, the ordering based on the failure rate does not perfectly match the ordering based on the use frequency. Also, we find that Extract Superclass, which is almost unused in practice (its share reported as less than 0.2%), is one of the most reliable refactorings; on the other hand, Extract Interface is the least reliable refactoring that has some number of real uses (reported as around 0.5%). We believe that the frequency of refactoring use should be a factor that aids in ranking the importance of bug reports.

6 Related Work

Testing refactoring engines requires input programs, which are rather complex test inputs. Programs can be represented as data structures such as abstract syntax trees (ASTs). Automated systematic generation of complex data structures has been proposed a while ago [2, 25], but only more recently Daniel et al. [7] proposed a systematic technique, called ASTGen, for testing refactoring engines. ASTGen requires the user to write imperative generators that can build parts of Java programs and offers a new approach to combine these generators. Given these generators, ASTGen systematically generates a large number of (small)

Java programs described with the generators. Although ASTGen exposed several bugs in Eclipse and NetBeans, it comes at the cost of writing the imperative generators, requires skillful users, and may not be adequate for describing some properties of complex inputs. We proposed UDITA [12], a non-deteministic language that enables the users to combine imperative generators with declarative filters to describe a set of complex test inputs in a concise way. Like ASTGen, UDITA requires rather sophisticated users. More recently Soares et al. [38, 39] follow an approach similar to UDITA and combine it with random testing to search for semantic changes introduced by refactorings. In contrast to these approaches, the approach presented in this paper introduces an end-to-end approach for testing refactoring engines on existing projects: applying refactorings on a number of existing projects, clustering failures, minimizing failing inputs, and detecting duplicate bugs. Our approach avoids the effort of writing test generators and increases the confidence that the bugs found are more important. However, our approach requires minimization of programs, which for now we perform manually; in the future, we plan to evaluate automated minimization approaches [27, 33, 50].

Applying refactorings on real programs has been explored by several researchers in different contexts. Spinellis [41] tested the Rename refactoring of his CScout refactoring engine on the Linux kernel source by systematically applying CScout to replace all identifiers with mechanically derived names and testing the correctness of the refactored code by checking that it compiles correctly. Thies and Steimann [44] tested two Eclipse refactorings, Move Class and Pull Up Method, by systematically applying them on existing open-source projects. Schäfer et al. [37] applied a few refactorings on more than million lines of open-source projects to investigate scalability of their refactoring implementation. Cinneide et al. [4] automatically applied refactorings on a number of projects as part of evaluating and comparing software metrics. Coker and Hafiz [5] tested three program transformations that fix C integer problems (signedness, overflow, underflow, and widthness problems) by applying these transformations on real C programs. In constrast, our paper evaluates an end-to-end approach for testing refactoring engines and evaluating their reliability on all refactorings in both Eclipse JDT and Eclipse CDT, and points out some key challenges in the process from applying refactorings to finding bugs, e.g., minimizing failing inputs and finding duplicate bugs.

Jagannath et al. [17] proposed clustering based on abstract messages of failures obtained by refactoring small corner-case programs. Our paper evaluates clustering on failures in real programs (there is no a priori reason to believe that a technique that works for small artificial corner cases also works well for real failures), applies it to the C language, and automates the abstraction of messages for JDT. Clustering has been also used for determining which program runs result in failures; most recently, Sun et al. [43] proposed a cost-sensitive strategy for inspecting clusters of program runs. In our approach, the outcome of each test is known after the execution of the test, and our end-to-end approach focuses on finding new, unique bugs.

Several projects have studied the usage of refactorings [6, 8, 28, 29, 45] and agreed that refactoring engines are underused. Most recently, Vakilian et al. [45], through a field study followed up by semi-structured interviews, investigated the reasons for low usage of refactorings and reported that the low usage is mostly due to unpredictability of the refactorings rather than their reliability. Our findings empirically confirm that the number of failures is not too high, but there is still a need for improvement and developing new infrastructure for building more reliable refactoring engines [15, 31, 34, 42, 45].

7 Conclusions

We presented a simple yet extremely effective approach to detect unique, real bugs in refactoring engines and to estimate their reliability. As opposed to previous techniques that generate input programs, our approach uses existing projects as inputs. As opposed to previous techniques that used existing projects as inputs for testing/evaluating refactorings, our approach identifies unique bugs using clustering, minimization, and finding duplicates. We applied our approach on testing Eclipse refactoring engines for Java and C, and we found and reported 77 new bugs for Java and 43 for C. We expect that bugs commonly found from real applications are more likely to be fixed than bugs discovered from artificially generated corner cases; in fact, the Eclipse developers already fixed 8 of the bugs we reported and confirmed 62 more as real bugs.

The main message of this paper is not that refactoring engines are buggy but that the proposed end-to-end approach works well to find these bugs. However, the approach also has challenges to be addressed in the future, e.g., automated minimization of programs and finding of duplicate bugs. While the paper focused on testing refactoring engines, we believe that the same approach can be used to test other aspects of IDEs that require programs/projects as test inputs, and to estimate their reliability on real projects.

Acknowledgments. We would like to thank Anirudh Balagopal for helping with the bug reports; Zack Coker, Sarfraz Khurshid, Rupak Majumdar, Aleksandar Milicevic, Andy Podgurski, and Friedrich Steimann for discussions about this work; and Caius Brindescu, Mihai Codoban, Yu Lin, Stas Negara, Cosmin Radoi, and Mohsen Vakilian for providing comments on the text. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0746856, CNS-0958199, CCF-1213091, and CCF-1217271.

References

1. Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S.: Duplicate bug reports considered harmful... really? In: ICSM, pp. 337–345 (2008)
2. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA, pp. 123–133 (2002)

3. CDT home page, http://www.eclipse.org/cdt
4. Cinnéide, M.Ó., Tratt, L., Harman, M., Counsell, S., Moghadam, I.H.: Experimental assessment of software metrics using automated refactoring. In: ESEM, pp. 49–58 (2012)
5. Coker, Z., Hafiz, M.: Program transformations to fix C integers. In: ICSE, pp. 792–801 (2013)
6. Counsell, S., Hassoun, Y., Loizou, G., Najjar, R.: Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. In: ISESE, pp. 288–296 (2006)
7. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: FSE, pp. 185–194 (2007)
8. Dig, D., Johnson, R.: How do APIs evolve? A story of refactoring. J. Softw. Maint. Evol. 18(2), 83–107 (2006)
9. Dig, D., Marrero, J., Ernst, M.D.: Refactoring sequential Java code for concurrency via concurrent libraries. In: ICSE, pp. 397–407 (2009)
10. Ebraert, P., D'Hondt, T.: Dynamic refactorings: Improving the program structure at run-time. In: RAM-SE, pp. 101–110 (2006)
11. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
12. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: ICSE, pp. 225–234 (2010)
13. GMP home page, http://gmplib.org
14. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: PLDI, pp. 206–215 (2008)
15. Hafiz, M., Overbey, J.: OpenRefactory/C: An infrastructure for developing program transformations for C programs. In: SPLASH, pp. 27–28 (2012)
16. Han, J.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers Inc. (2005)
17. Jagannath, V., Lee, Y.Y., Daniel, B., Marinov, D.: Reducing the costs of bounded-exhaustive testing. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 171–185. Springer, Heidelberg (2009)
18. JDT home page, http://www.eclipse.org/jdt
19. JPF home page, http://babelfish.arc.nasa.gov/trac/jpf
20. JUnit home page, http://junit.org
21. Eclipse Juno home page, http://www.eclipse.org/juno
22. libpng home page, http://www.libpng.org/pub/png/libpng.html
23. Apache log4j home page, http://logging.apache.org/log4j/2.x
24. Lucene home page, http://lucene.apache.org
25. Marinov, D., Khurshid, S.: TestEra: A novel framework for testing Java programs. In: ASE, pp. 22–31 (2001)
26. Commons Math home page, http://commons.apache.org/math
27. Misherghi, G., Su, Z.: HDD: Hierarchical delta debugging. In: ICSE, pp. 142–151 (2006)
28. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: ICSE, pp. 287–297 (2009)
29. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A comparative study of manual and automated refactorings. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 552–576. Springer, Heidelberg (2013)
30. OpenRefactory/C - A refactoring infrastructure for C, http://openrefactory.org

31. Overbey, J.L.: A Toolkit For Constructing Refactoring Engines. PhD thesis, University of Illinois at Urbana Champaign (2011)
32. Refactoring actions home page, http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.user/concepts/concept-refactoring.htm
33. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: PLDI, pp. 335–346 (2012)
34. Schäfer, M.: Specification, Implementation and Verification of Refactorings. PhD thesis, Oxford University Computing Laboratory (2010)
35. Schäfer, M., Ekman, T., de Moor, O.: Sound and extensible renaming for Java. In: OOPSLA, pp. 277–294 (2008)
36. Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Refactoring Java programs for flexible locking. In: ICSE, pp. 71–80 (2011)
37. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A comprehensive approach to naming and accessibility in refactoring Java programs. IEEE Trans. Soft. Eng. 38(6), 1233–1257 (2012)
38. Soares, G.: Making program refactoring safer. In: ICSE, pp. 521–522 (2010)
39. Soares, G., Catao, B., Varjao, C., Aguiar, S., Gheyi, R., Massoni, T.: Analyzing refactorings on software repositories. In: SBES, pp. 164–173 (2011)
40. Song, Y., Wang, X., Xie, T., Zhang, L., Mei, H.: JDF: Detecting duplicate bug reports in Jazz. In: ICSE, pp. 315–316 (2010)
41. Spinellis, D.: CScout: A refactoring browser for C. Sci. of Comp. Prog. 75(4), 216–231 (2010)
42. 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)
43. Sun, B., Shu, G., Podgurski, A., Ray, S.: CARIAL: Cost-aware software reliability improvement with active learning. In: ICST, pp. 360–369 (2012)
44. Thies, A., Steimann, F.: Systematic testing of refactoring tools. In: AST (poster) (2010), http://www.fernuni-hagen.de/ps/prjs/rtt/rtt_poster.pdf
45. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: ICSE, pp. 233–243 (2012)
46. Visual Assist X home page, http://www.wholetomato.com
47. Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information. In: ICSE, pp. 461–470 (2008)
48. Wloka, J., Hirschfeld, R., Hänsel, J.: Tool-supported refactoring of aspect-oriented programs. In: AOSD, pp. 132–143 (2008)
49. XRefactory home page, http://www.xref.sk/xrefactory/main.html
50. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Soft. Eng. 28(2), 183–200 (2002)
51. zlib home page, http://www.zlib.net

Simple Profile Rectifications Go a Long Way

Statistically Exploring and Alleviating the Effects of Sampling Errors for Program Optimizations

Bo Wu[1], Mingzhou Zhou[1], Xipeng Shen[1],
Yaoqing Gao[2], Raul Silvera[2], and Graham Yiu[2]

[1] Computer Science Department, The College of William and Mary, VA, USA
[2] IBM Toronto Lab., Toronto, Canada

Abstract. Feedback-driven program optimization (FDO) is common in modern compilers, including Just-In-Time compilers increasingly adopted for object-oriented or scripting languages. This paper describes a systematic study in understanding and alleviating the effects of sampling errors on the usefulness of the obtained profiles for FDO. Taking a statistical approach, it offers a series of counter-intuitive findings, and identifies two kinds of profile errors that affect FDO critically, namely zero-count errors and inconsistency errors. It further proposes statistical profile rectification, a simple approach to correcting profiling errors by leveraging statistical patterns in a profile. Experiments show that the simple approach enhances the effectiveness of sampled profile-based FDO dramatically, increasing the average FDO speedup from 1.16X to 1.3X, around 92% of what full profiles can yield.

1 Introduction

Feedback-Driven Optimization (FDO) is a technique modern compilers use to optimize programs based on some profiles of the program dynamic behaviors, such as the frequencies of basic blocks and function invocations. FDO has proven effective for improving program performance. It has been part of most commercial compilers (e.g., IBM XLC, Intel ICC) of traditional imperative languages. It is also essential for modern languages with a managed environment (e.g., Java, Javascript, Python.) The runtime engines of these languages nowadays commonly employ Just-In-Time (JIT) compilers to complement interpreters for producing code with a good performance. Most optimizations by the JIT compilers are FDO: They make optimization decisions based on some profiles the runtime engine collects throughout the current execution. With JIT compilation becoming popular for modern languages, the effectiveness of FDO is becoming increasingly important for the efficiency of modern computing.

The profiles used for FDO are often collected through sampling, because collecting full profiles requires some detailed instrumentations and hence incurs lots of overhead. It is especially true for JIT-based languages, as for them, profile collections happen usually on the fly.

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 654–678, 2013.

Sampling unavoidably introduces some inaccuracy into the collected profiles, which may in turn impairs the effectiveness of FDO. There have been some studies on reducing biases in the sampling schemes used in Java runtime systems [16], and some proposals of improved sampling schemes—such as bursty sampling [5, 6, 11, 15]. Although these techniques can improve the profiles quality in a certain degree, the speedup by FDO on the sampled profiles still has a substantial gap from what it produces on full profiles, as Figure 1 shows.

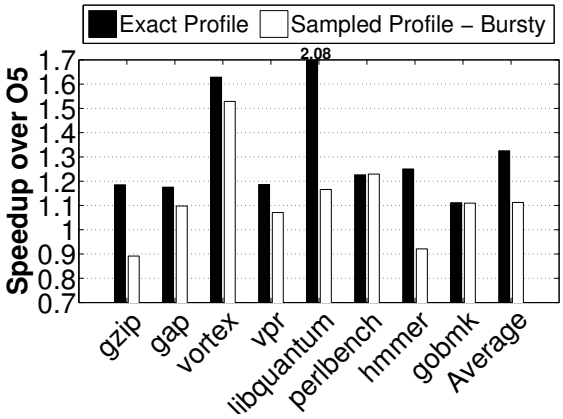


Fig. 1. The speedups (compared to static compilation with the highest-level optimizations enabled) produced by the FDO of IBM XLC (v12.1) compiler, when sampled (bursty at rate of 5%) and full profiles are used respectively

In this work, we attack the problem from a different angle: Rather than refining sampling schemes, we try to rectify the errors in a profile after it is collected. It is called *profile rectification*.

For profile rectification to work, we must answer some fundamental questions: What are the relations between sampling rates, accuracy of the collected profile, and its usefulness for FDO? How do the errors in a profile influence the optimizations? And how to rectify the critical errors? Answers to these questions are essential for guiding the directions of profile rectification. But to the best of our knowledge, none of these questions have been systematically studied on modern compilers and programming systems.

Prompted by these open questions, we conduct two investigations in this work. First, as Section 2 will show, we design a set of systematic measurements to reveal the statistical correlations among sampling rates, profile accuracy, and the corresponding FDO benefits. To avoid biases in the analysis, we conduct 7680 runs, which cover seven most important factors in four levels, including the usage of two mature compilers, two sampling methods with six sampling rates for each, two platforms, eight SPEC benchmarks with some non-trivial FDO potential, four inputs per benchmark, and ten repetitions for each setting.

The systematic measurements reveal some counter-intuitive observations. It is commonly perceived that a higher sampling rate tends to give more accurate profiles, which in turn would help FDO produce code that has a better performance. However, the measurement suggests that even though a higher bursty sampling rate gives a more accurate profile in general, the perception does not hold for uniform sampling, which samples once in a fixed time period[1]. Moreover, the results show that for both types of sampling methods, when a more accurate profile is given to the FDO, it often fails to produce code with a better performance. In other words, in the sampling rate range, profile accuracy does *not* have an apparent correlation with the FDO speedups.

The surprising observations prompt our two-fold investigation (Section 3). We conduct a deep analysis of the influence cast on FDO by various types of sampling errors. We identify two types of errors that have some important influence. The first is *zero-count errors*, which refer to the case when a counter in sampled profile equals zero but its value in the full profile is not. The second is *inconsistency errors*, which refer to the case when two counters in the same function have different values in the sampled profile but have the same value in the full profile. Our analysis shows that although these two types of errors do not affect the overall profile accuracy much, they alter the behaviors of the FDO dramatically. Based on the findings, we propose to rectify the two types of errors through exploitation of the statistical patterns in profile counters derived from some training runs. The rectification, although being simple, turns out to boost the usefulness of the sampled profiles for FDO significantly, increasing the average speedup from 1.16X to 1.3X, around 92% of what full profiles can yield.

There are two prior studies on correcting sampling errors in a profile for FDO [10, 14]. They both apply a minimum cost circulation algorithm to the control flow graphs of a program, hence subject to the static constraints in the program (detailed in Section 4). Our rectification method is distinct in exploiting *statistical* patterns shown in *dynamic* profiles. The exploited dynamic patterns are essential for correcting the two kinds of critical errors.

In summary, this work makes three main contributions.

- **Correlations** It provides the first study in modern systems that systematically uncovers the correlations among sampling rates, profile accuracy, and the usefulness for FDO. Through a comprehensive measurement using contemporary sampling techniques and compilers, the study produces some findings contrary to common perceptions.
- **Influence of Errors** It offers a set of novel insights on sampling and its influence on FDO:
  - The zero-count and inconsistency errors in sample profiles hurt FDO substantially, despite their modest influence on the overall profile accuracy.
  - Uniform sampling not only underperforms bursty sampling, but shows a weak correlation between sampling rate and profile accuracy, which

[1] The "time" here could be wall-clock time or logical time (e.g., a number of instructions or basic blocks).

reinforces the superiority of bursty sampling over uniform sampling for FDO.
  - Commonly defined accuracy, either weighted or unweighted, fails to quantify the actual quality of a profile for FDO.
- **Statistical Profile Rectification** To our best knowledge, this is the first work showing that simple statistical profile rectification can dramatically enhance the usefulness of a profile for FDO.

The rest of the paper is organized as follows. In Section 2, we first briefly introduce the two FDO compilation systems, the two sampling methods, and some other experiment settings. Then we present the findings of the correlations. In Section 3, we present a deep analysis of the sampling errors, and describe the simple profile rectification method. We discuss some related work in Section 4 and then conclude the paper with a summary.

2 Counter-Intuitive Correlations

This section starts with an introduction to the FDO in the two compilers we use. It then presents the design of the empirical measurements and reports the findings on the correlations among sampling, profiles, and their usefulness.

2.1 Background on FDO

FDO is part of many modern compilers. The implementations of FDO in different compilers may differ in what set of optimizations they contain, but mostly follow a similar high-level design. We briefly describe the way FDO works in a mature commercial compiler, IBM XLC, as follows.

To enable FDO, two stages of compilations are necessary. For XLC, in the first stage, the compiler must be invoked with a special option ("-qpdf1"). With that option, the compiler instruments the program with some monitoring instructions. An execution of the generated executable will produce a profile of that execution. In the second stage, the compiler is invoked again with another special option ("-qpdf2"). In this round of compilation, the compiler enables FDO, which reads the profile and produces an optimized executable.

JIT follows a similar two-stage scheme. The main difference from XLC-like offline compilers is just that JIT invokes the two rounds of compilation implicitly at runtime. For instance, in a Java virtual machine JikesRVM [1], the first-time compilation of a newly loaded Java method inserts some *yielding points* into the generated code, which help to sample runtime behaviors of the method in its execution. If the Java runtime decides to recompile that method later in that run, its JIT compiler will use the sampled profile to do a FDO on that method.

As the profiles capture some runtime behaviors (e.g., the hotness of a function or basic block), they can provide the optimizers some hints that static code analysis is unable to provide. FDO heavily exploits those hints to enhance code layout, inline functions, and so on. An inaccurate profile may hence mislead FDO into making wrong optimization decisions.

2.2 Experimental Design

We design a set of experiments to empirically measure the relations among sampling rate, profile accuracy, and the influence on the effectiveness of FDO. In the design, we carefully cover seven dimensions that are closely relevant to the relations to minimize the bias in the measurement. They fall into four levels. Table 1 shows the dimensions, the number of optional values in each dimension (the "Variations" column) and a brief description of the optional values. We explain these dimensions in more details as follows.

Table 1. Dimensions covered in the experiment design

Levels	Dimensions	Variations	Description
workload	benchmarks	8	from SPEC CPU2000 and CPU2006
	inputs	4	*1 train input, 3 ref inputs*
system	compilers	2	XLC, GCC
	platforms	2	Intel Xeon & IBM POWER7
sampling	methods	2	Bursty, Uniform
	frequencies	12	6 for Bursty, 6 for Uniform
noise avoidance	repetitions	10	number of repetitive runs per setting

Benchmarks and Inputs. Given that the focus of this work is on FDO, when choosing benchmarks, we concentrate on those that exhibit some non-trivial speedups when FDO is applied. Meanwhile, our current infrastructure works on C programs only. Among the programs in SPEC CPU2000 and CPU2006 [2], we find eight of them meeting both criteria, as listed in Table 2. All these benchmarks are integer programs, and have complex control flows and a large number of functions, posing challenges for static analysis and hence exhibiting good potential for FDO. As Figure 1 shows, these benchmarks show an average 1.33X speedup when FDO is applied (on the exact profiles of the execution inputs) compared to their performance through static compilations using XLC.

For each program, besides including both its *train* and *ref* inputs coming by default with the benchmark suite, when necessary, we collect or create two extra representative inputs by searching for the real usage of their original applications or reading the source code. The extra inputs are used in the experiments described in Section 3 for examining the stableness of profile value patterns across different inputs. For FDO, the profiles are collected on the train input and evaluated on the ref input.

Compilers. For compilers, there are two possible choices: some mature highly polished offline compilers, or some JIT compilers. We choose the former for the following reasons. First, as Section 2.1 mentions, the FDO in JIT compilers is triggered by some decisions made by the runtime engine and is hard to control and hence experiment with. Using it would add more noise into the measurement.

Table 2. Benchmarks and FDO speedup from exact profiles over O5 compilation

Program	Benchmark Suite	Description	FDO speedup
gzip	CPU2000	Compression	1.19X
gap	CPU2000	Group Theory, Interpreter	1.18X
vortex	CPU2000	Object-oriented Database	1.63X
vpr	CPU2000	FPGA Circuit Placement and Routing	1.19X
libquantum	CPU2006	Quantum Computing	2.09X
perlbench	CPU2006	Perl Interpreter	1.22X
hmmer	CPU2006	Search Gene Sequence	1.25X
gobmk	CPU2006	Artificial Intelligence	1.11X

Second, JIT compilers are not as mature as offline compilers. The FDO in offline compilers has been developed for decades and has reached a quite stable state. Compared to JIT, the implementation in offline compilers usually tap into the potential of FDO to a much higher degree, because for the tradeoff between runtime overhead and code quality, what optimizations JIT compilers should include is still an active research topic yet to settle. For all these reasons, using current JIT compilers is hard to uncover the principled relations between profile accuracy and FDO.

We select the recent versions of XLC (v12.1) and GCC (v4.6.2) as our compilers. Both compilers have been developed for more than a decade. The former is the main commercial compiler of IBM for C and C++, and shares the core with many other IBM compilers for other languages. Its FDO is sophisticatedly polished by a large compiler team for many years, able to exploit profiles to conduct a number of advanced intra-procedure and inter-procedure optimizations. GCC is a result of the many years of efforts by the open-source community. Its performance has been shown to get close to commercial compilers in many cases. Its FDO component has also been developed for quite a while. We use both compilers for this study to examine the influence of different FDO implementations on the studied relations.

In the instrumentation stage of XLC, each procedure's control flow graph is explored to find out some straight lines of basic blocks that must have the same counts. Only the first basic block of a straight line needs to be instrumented to collect access frequency. A mapping data structure records which functions are invoked in which basic blocks based on static call graphs. During recompilation, function calling frequencies and control flow branch probabilities are inferred from basic block counters and the mapping, serving as hints for the FDO. A similar implementation scheme is shown in GCC, although differences exist in the set of optimizations they include and how those optimizations are implemented.

Platforms. We run XLC-related experiments on an IBM Power7 machine, which has the AIX 7 operating system installed. We conduct the GCC-related experiments on a machine equipped with Intel Xeon W3550, running a OpenSUSE Linux, version 12.1.

Sampling Methods and Frequencies. We experiment with two sampling methods. The first is uniform sampling, which is the most commonly used sampling method. It tries to get a sample after a given time interval. For example, when being applied to collect basic block frequency profiles, the runtime sampler checks which instruction is being executed and finds out which basic block that instruction belongs to after a given time interval; it then increases the counter corresponding to the basic block by one. The second sampling method is bursty sampling. In this method, there are two predefined parameters, the *execution period length* τ_e and the *profiling period length* τ_p. The runtime switches between normal execution and profiled execution periodically. During an execution, after a τ_e-long period of normal execution, the runtime switches the execution to a fully instrumented version and runs that version for a τ_p-long period of time to collect some profiles, and then switches back to normal execution. The back-and-forth switching continues throughout the program execution.

Since we use static compilers, which do not have a runtime sampling system, we simulate the two sampling methods. For uniform sampling, we assume that each instruction's execution takes equal time and thus has the same probability to be sampled. The full profiles are processed to obtain the sample profiles. For bursty sampling, we modify the instrumentation, so one execution directly produces one sample profile.

Previous studies have shown that the bursty sampling, although being more complicated to implement, can often produce a more accurate profile than the uniform sampling does at the same sampling rate [5]. Bursty sampling has been implemented in some runtime systems, such as Jikes RVM [6]. Using both sampling methods helps us examine the influence of different sampling schemes on the relations between profile accuracy and FDO.

We experiment with six sampling rates for each of the sampling methods. These rates subsume the typical range of sampling rates used in practical systems. The sampling rate of uniform sampling is determined by a single parameter, the sampling period length. Because the bursty sampling has two parameters, the execution period length τ_e and profiling period length τ_p, each of its sampling rates is represented with the ratio of a pair, $\tau_p/(\tau_e+\tau_p)$. Specifically, for uniform sampling, the used sampling rates are $100, 1000, 10,000, 50,000, 100,000, 500,000$; for bursty sampling, the rates are $1/1000, 10/1000, 50/1000, 100/1000, 200/1000, 400/1000$.

Time Measurement. In all runs, the highest optimization level is enabled. We see some minor fluctuations in the execution times of multiple runs in the same setting. But still to minimize the influence of random noise, we repeat each run for 10 times and use their average time for comparison.

2.3 Measurements and Findings

The coverage of the various factors leads to 7680 runs in total. This subsection presents the findings we have obtained from these measurements. But before that, we first explain some metrics we use to quantify profile accuracy and correlations.

Accuracy Metrics. Let B_i represent the exact profile (or called full profile) of a run on input i. Exact profiles can be obtained through a full profiling. Let SP'_i be the profile obtained by sampling. Before comparing the two profiles, we multiply each counter in SP'_i by the ratio between the sum of the counters in B_i and that in SP'_i so that the two profiles are at the same scale for comparison. We denote the scaled sampled profile with SP_i. We use $SP_i[j]$ and $B_i[j]$ to denote the counter values of the jth item (e.g., basic block frequency counter) in the sampled and exact profiles, respectively. For the purpose of explanation, we use basic block frequency profiles as our example in the following discussion. In such a profile, each item corresponds to the frequency of a basic block being accessed.

The definition of the accuracy should quantify the similarity between SP_i and B_i. Following common practices, we define the *accuracy* (Acc) of a basic block counter as follows:

$$Acc_i[j] = 1 - \frac{|SP_i[j] - B_i[j]|}{max(SP_i[j], B_i[j])}$$

The use of *max* in the denominator is to normalize the accuracy to the range of [0, 1]. We use two definitions for the overall accuracy of a profile. An *unweighted accuracy* (UAcc) is just an arithmetic average of all basic blocks' accuracies. It treats each basic block equally. A *weighted accuracy* (*WAcc*) of a profile is a weighted average as follows:

$$WAcc_i = \sum_j Acc_i[j] \times \frac{B_i[j]}{\sum_j B_i[j]}$$

where, the weights are proportional to the significance of a basic block in the program in terms of its access frequency.

Correlation Metrics. Among the different variations of commonly used correlations metrics, we find the *Spearman's rank correlation coefficient* (called *rank coefficient* in short) suiting our needs. Let X and Y represent two ordered lists of values. For instance, in our experiment, we set X to be the list of accuracies of a number of sampled profiles, and Y be the list of speedups brought by FDO based on those profiles. Elements in both X and Y are sorted in an ascending order of their values. The position of an element in the ordered list is called the *rank* of that element. The rank coefficient is defined as follows:

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \quad ,$$

where, $x[i]$ and $y[i]$ are the ranks of $X[i]$ and $Y[i]$ in X and Y respectively.

For example, consider the following case. We have four sampled profiles of a program. Their accuracies are shown as the X column and their FDO speedups as the Y column in Table 3. The ranks of each profile in the two lists are shown in the two rightmost columns. In the definition of the correlation metric, x_i and y_i represent the ranks; when $i = 2$, they equal 1 and 4 respectively. The symbols, $\bar{x}$ and $\bar{y}$ in the rank coefficient formular, represent the average of the ranks in

X and Y. They both equal 2.5 in this example. The rank coefficient between X and Y is -0.4.

Table 3. Example for illustration of rank coefficient

profile-ID	X	Y	X's rank	Y's rank
1	0.89	1.1	3	3
2	0.93	1.08	1	4
3	0.90	1.2	2	1
4	0.86	1.15	4	2

Recall that the questions we try to answer are whether a higher sampling rate leads to a more accurate profiles and hence more benefits from FDO. The rank coefficient fits our needs as it assesses how well the relationship of two variables fits in a monotonic function. In comparison, the standard Pearson coefficient measures whether two variables form a linear relation, which is a property unnecessarily stronger than what we need.

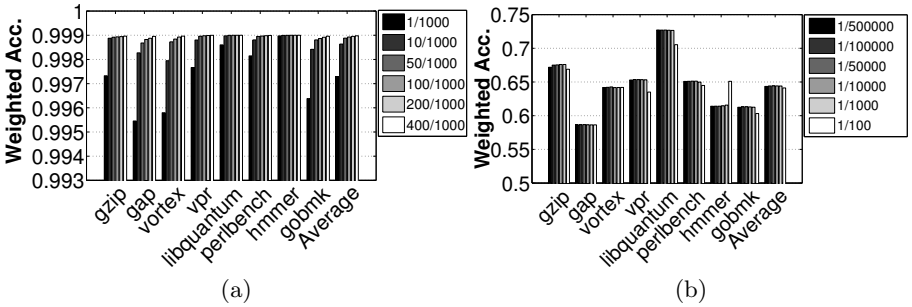


Fig. 2. (a) Weighted accuracy of bursty sampling. (b) Weighted accuracy of uniform sampling.

The value of a rank coefficient is always between -1 and 1, with a value close to 1 implying a strong co-increasing relation between X and Y, and a value close to -1 implying that the two variables' values are taking an opposite trend.

Sampling Rate and Profile Accuracy. Figures 2 and 3 report the weighted and unweighted accuracies of the sampled profiles of all benchmarks when different sampling rates are used. The profiles for the bursty sampling have a close-to-perfect weighted accuracy across all sampling rates, while the profiles for the uniform sampling have an average 64% accuracy. The intuition behind the large accuracy disparity is that because each time the uniform sampling checks only one instruction, a larger basic block gets some larger chance to be sampled than a smaller basic block does if the two blocks actually have the same frequencies

of being executed. The issue is less serious in bursty sampling. For bursty sampling, block size may cast some influence on which block the sampling period starts from, but the influence is much weaker to the overall accuracy because within a sampling period, the size of a basic block less affects the chance for it to get sampled. These results echo some previous observations on the two sampling methods [5, 11]. The unweighted accuracy difference is smaller, but bursty sampling still outperforms uniform sampling in general.

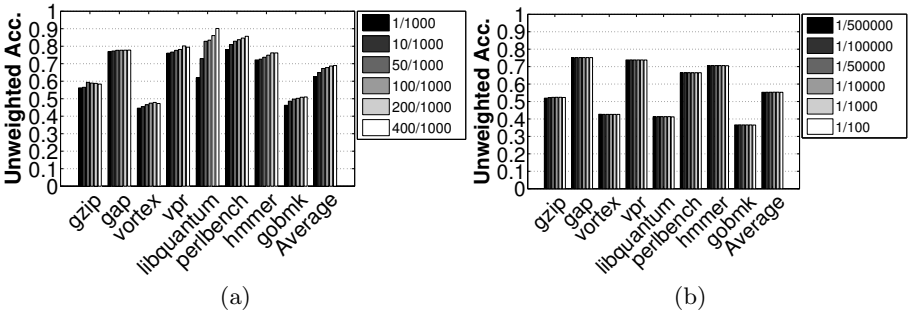


Fig. 3. (a) Unweighted accuracy of bursty sampling. (b) Unweighted accuracy of uniform sampling.

Table 4 provides the rank coefficients between sampling rate and profile accuracies. When weighted accuracy is used, the coefficients are all 1 for bursty sampling, indicating the very strong correlation between sampling frequency and profile accuracy. In other words, the profile accuracy will definitely increase when we use a higher sampling rate. When unweighted accuracy is used, the correlations are slightly lower, but still close to one for most programs.

Uniform sampling shows much weaker correlations with profile accuracy. The average of the rank coefficient is only -0.22 when weighted accuracy is used. Two programs, *hmmer* and *libquantum*, are exceptions. For *hmmer*, a higher sampling rate leads to more accurate profiles, while for *libquantum*, the trend is the opposite. Overall, a higher rate of uniform sampling does not lead to a more accurate profile. The reason for the weak correlations comes from the same source (the effects of basic block sizes) for the low profile accuracy mentioned earlier in this section. To further understand the severity of the effects, we extend the sampling rate to some large values (20%, 40%, 80%) that are rarely used in actual runtime sampling. The results show that even at such a level of sampling rates, the correlations are no much stronger than what Table 4 has shown.

Table 4. Rank correlation coefficients between sampling frequency and profile accuracy

Program	Weighted Bursty	Unweighted Bursty	Weighted Uniform	Unweighted Uniform
gzip	1	0.43	0.09	0.66
gap	1	1	-0.6	0.94
vortex	1	0.83	-0.03	-0.94
vpr	1	0.94	-0.43	-0.37
libquantum	1	1	-0.94	0.086
perlbench	1	1	-0.49	-0.6
hmmer	1	0.94	1	-0.66
gobmk	1	1	-0.37	-0.43
Median	1	0.97	-0.4	-0.4

These results provide two insights. First, they confirm and further reinforce that bursty sampling is more suitable for program profiling than uniform sampling. Second, the weak correlations of uniform sampling suggest that the shortcoming of uniform sampling for program profiling is deeply inherent in the method, and can hardly be overcome by an increase in sampling rate. Given that uniform sampling is still the most commonly used runtime profiling method in today's systems, these insights hopefully will prompt developers to revisit the sampling methods they select.

Profile Accuracy and FDO Benefits. Figure 4 reports the speedups FDO produces on the full profiles and profiles collected through bursty sampling at three sampling rates. Recall that for bursty sampling, higher sampling rates always lead to more accurate profiles. However, the bars in Figure 4 show a quite irregular pattern in the speedups as sampling rate increases. While *gap*, *vortex* and *perlbench* follow the intuitive trend of benefiting more from more accurate profiles, all other benchmarks show an opposite trend sometimes—degraded performance from more accurate profiles. The extreme case on *libquantum* even shows 75% more speedup from the lowest sampling rate than from the highest sampling rate. Overall, the FDO shows the best effectiveness on the exact profiles, 17% more speedups than on the best sampled profiles.

Table 5 reports the rank coefficients between profile accuracies and the speedups. No benchmarks have near 1 correlation coefficient. Only two benchmarks (*vortex, perlbench*) have coefficients larger than 0.8 on bursty sampling when weighted accuracy is used. So for them, higher bursty sampling rates are likely to bring better optimizations. But for most benchmarks, there is only weak or no correlation between profile accuracy and the usefulness for FDO. The program *gap* even has a coefficient of -0.85 on uniform sampling, indicating a largely monotonic decreasing relation between profile accuracy and usefulness for FDO.

Short Summary. Current FDO optimization systems are constructed mostly on a common perception that larger sampling rates tend to lead to better performance. This section debunks the intuition by first showing that higher sampling

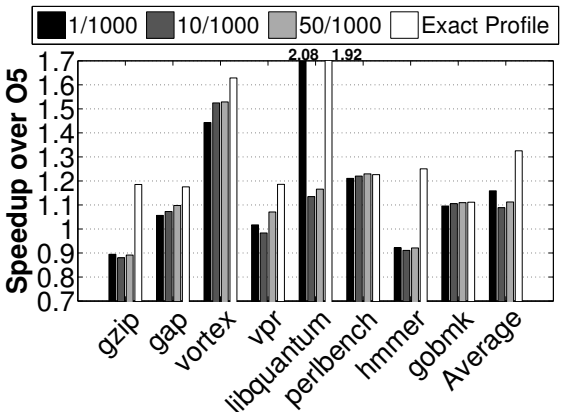


Fig. 4. Speedup comparison between sampled profiles of three sampling rates(1/1000, 10/1000 and 50/1000) and exact profiles

frequency does not necessarily give us more accurate profiles, and it depends on the sampling method used. This indicates that we should be more careful about the design of sampler. More surprisingly, we show that there are very weak correlations between the accuracy of a profile and its usefulness for FDO, no matter which sampling method is used. It does not mean that we can just feed the compiler with randomly generated profiles for good FDO-driven performance. As results show, the best performance mostly still come from the exact profiles for most benchmarks. The findings suggest that current understanding to how profiling errors influence FDO is preliminary; some deep analysis into the results are necessary, as given in the next section.

Table 5. Rank Correlation coefficients between profile accuracy and performance

Program	Weighted Bursty	Unweighted Bursty	Weighted Uniform	Unweighted Uniform
gzip	-0.14	-0.29	-0.15	0.58
gap	0.75	0.75	-0.85	0.34
vortex	0.88	0.59	0.08	-0.8
vpr	0.62	0.79	-0.01	-0.07
libquantum	-0.08	-0.08	-0.5	0.16
perlbench	0.82	0.82	0.51	0.63
hmmer	0.47	0.41	0.11	-0.76
gobmk	0.41	0.41	-0.42	-0.59
Median	0.55	0.5	-0.08	0.05

3 Demystification and Profile Rectification

The previous section showed that for most benchmarks, there exists only very weak correlation between profile accuracy and its usefulness for FDO. However,

we observe that sampled profiles do not perform as well as exact profiles, which means sampling errors do play an important role. After analyzing the influence of various types of errors, we identify two kinds of sampling errors that critically affect the FDO benefits: *zero-count errors* and *inconsistency errors*. In this section, we first present some analysis results on how these two kinds of errors impair the effectiveness of FDO, and then show that both types of errors can be fixed through a simple profile rectification, and finally report the dramatic speedup increment the rectification helps FDO generate.

3.1 Deep Analysis on Profile Errors

Zero-count Errors. The first type of errors is zero-count errors, referring to the case when a counter in a sampled profile equals zero but its value in the full profile is not. For the purpose of explanation, we will concentrate our discussion on basic block frequency profiles.

Sampling, by nature, misses some parts of a program execution. But basic blocks that have a small value in the exact profile are especially easy to be missed completely by the sampler. Given the 20-80 rule (i.e., commonly 20% of a program is responsible for about 80% of its execution), most basic blocks are relatively cold, and hence have some good probabilities to get missed by the sampler, causing zero-count errors.

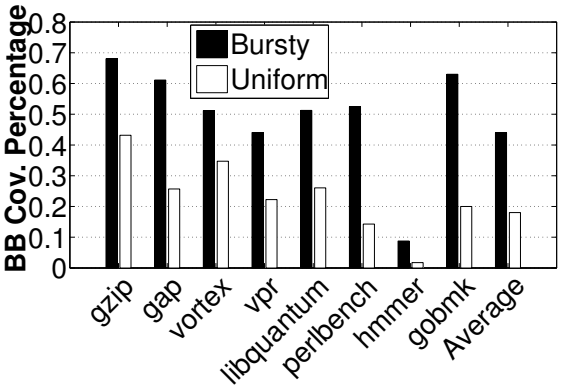


Fig. 5. Basic block coverage comparison between exact profiles and sampled profiles

Figure 5 shows the basic block coverage of the sampled profiles[2]. The *coverage* is defined as the percentage of the non-zero counters in the exact profile that also have non-zero values in the sampled profile. This metric shows how well the sampled profile represents the coverage pattern of the exact profile. We observe an average of 56% basic block coverage reduction by bursty sampling. In the

[2] Without noting, the results in this and following figures are similar across sampling rates, and the results at the lowest sampling rate is used.

worst case shown on *hmmer*, more than 92% of basic blocks are completely missed by the sampler. The coverage by uniform sampling is even worse, only 18%.

Through a detailed analysis of the influence of zero-count errors on the various optimizations in FDO, we find that two optimizations, function inlining and loop optimizations, are influenced the most. As Section 2.2 has mentioned, function calling frequencies are inferred from basic block frequencies in XLC. If the basic block containing a function call has zero frequency, the recompilation totally ignores the corresponding call edge. If all basic blocks invoking a function have zero frequencies, all the profile information of that function is ignored. This implies that the counter values of calling basic blocks play an important role in making inlining decisions, which is supported by Figure 6. It reports the number of function inlinings the FDO does when it uses a sampled profile, normalized by the number when it uses the full profile. On average, the zero-count errors cause the FDO to miss 79% inlining opportunities.

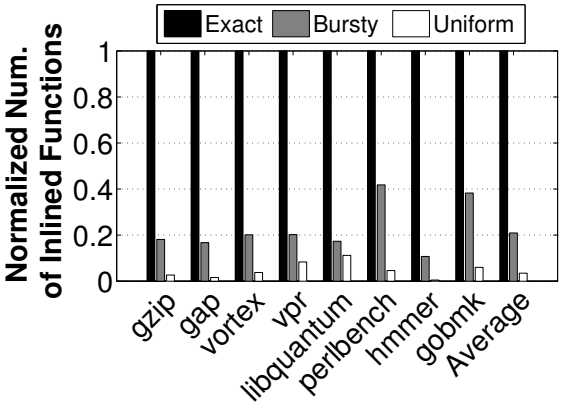


Fig. 6. Normalized number of inlined functions

The second type of transformation, loop optimizations, also leverages profile information heavily. For example, in XLC, the iteration counts of all loops are calculated through the basic block counters in loop body and that of the loop preheader. If a loop's preheader's counter value is 0, its iteration count is annotated as "unknown". Since iteration count is one of the most important parameter in most loop transformations (e.g., loop versioning, loop unrolling, etc.), false information on loop iteration count may seriously impair the transformation quality. However, due to the fact that loop preheader is usually executed much less frequently than its corresponding loop body, it is quite possible that although the sampler obtains a reasonable profile of the loop body, it can not take advantage of it because of a zero counter value of the loop preheader. Figure 7 shows the percentage of loops which have "unknown" iteration counts when the sampled profile is used, while non-zero iteration counter when the exact profile is used.

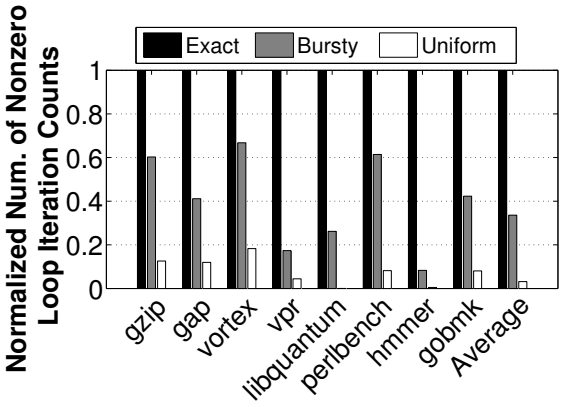


Fig. 7. Normalized number of loops having non-zero iteration counts

On average, only 33% of loops derived their iteration count information from the sampled profile. This percentage dropped to 3% for uniformly sampled profiles.

We now try to rectify the zero-count errors to show their impact on performance. Figure 8 reports the results when the zero counters in the sampled profile are set to the values of their counterparts in the exact profile. That is, this rectification replaces the zero counters in the sampled profile with perfect information and hence completely removes the zero-count errors. We observe an increase of 10% and 19% performance improvement for bursty and uniform sampling respectively. It echoes the results of basic block coverage and inlining decision difference, and shows that zero-count errors are one of the main sources leading to reduced FDO benefit.

Inconsistency Errors. Zero-count errors mainly happen on cold events, while inconsistency errors also happen on warm or hot events. An inconsistency error refers to the case when two counters of two basic blocks in the same function have different values in the sampled profile but have the same value in the full profile.

For hot events, both sampling methods can get pretty good approximation of their values, which is reflected by the very high weighted accuracy reported in Section 2. However, a decent approximation cannot prevent inconsistency errors from happening.

To help quantify the amount of inconsistency errors in the sampled profiles, we introduce a concept called *consistency score*. Let G represent a group of basic blocks in an exact profile that have the same counter values, and G' be the largest subset of G that have identical counter values in a sampled profile. The *consistency score* of G in the sampled profile is $\frac{|G'|}{|G|}$. So the score must fall between 0 and 1; the higher it is, the better is the consistency preserved in the sampled profile. The overall consistency score of a sampled profile is just the average of the consistency scores of all the consistent groups in the corresponding exact profile.

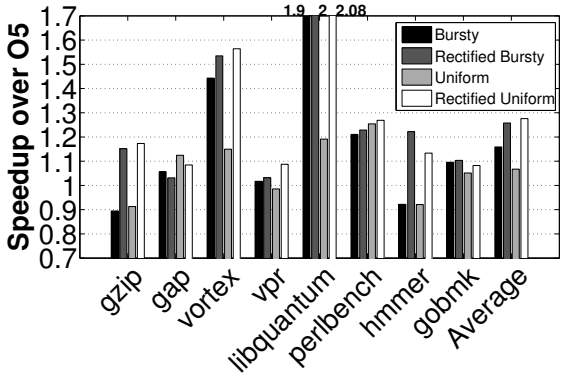


Fig. 8. Performance improvement after fixing zero-count errors with exact profiles

Figure 9 reports the consistency scores of the sampled profiles. On average, the profiles have consistency scores of 0.47 and 0.01 for bursty and uniform sampling respectively, suggesting that the sampling methods cannot preserve the consistency relation among counters well.

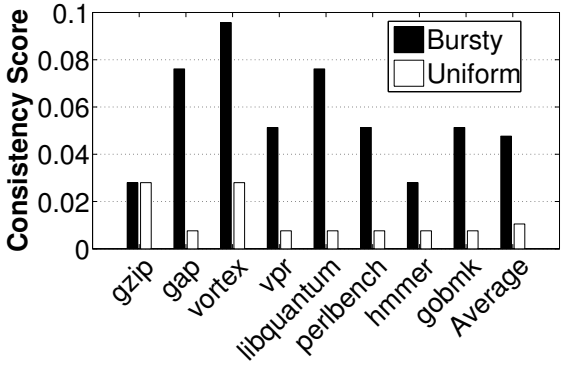


Fig. 9. Consistency scores of all benchmarks

We study the potential performance gain by leveraging exact profiles to help rectify the inconsistency errors in the sampled profiles. We identify all the basic block groups of each function in the exact profile that have the same counter value. Then, we set the counters in each group of the sampled profiles to their average. In this way, we maintain the equality relationship without changing the sampled profile's accuracy much. Figure 10 shows an improvement of up to 34% for uniform sampling on *vortex*, demonstrating the large potential of fixing inconsistent basic block counters. For bursty sampling, we have an outlier *libquantum*, for which the rectification degrades the performance by 90%. A plausible reason is that as the rectification is applied to its inconsistency errors only, the rectified profile somehow forms some serious conflict with the zero-count

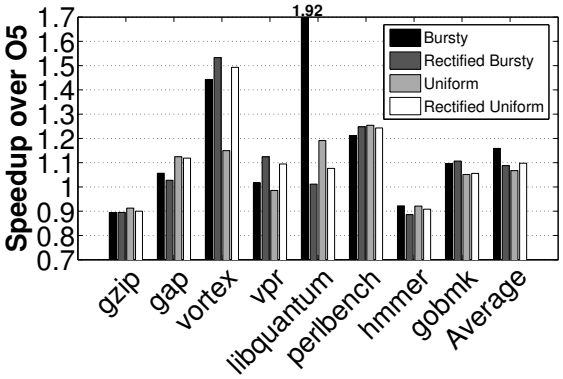


Fig. 10. Performance improvement after fixing inconsistency errors with exact profiles

errors remaining in the profile. Such an inference comes from an observation the next subsection (Figure 13) will show: The degradation is completely reversed when zero-count errors are also fixed.

A detailed analysis shows that the primary influence of the inconsistency errors is also on function inlining. The XLC compiler makes inlining decisions based on function hotness, size, and other factors. It first chooses the functions whose calling frequencies exceed a threshold to inline. If two functions have the same frequency, it chooses the smaller one to inline. So consider two functions, A and B (assuming A is much larger than B), have the same frequency in the exact profile but different in the sampled one (A has a larger frequency than B). The inconsistency error may hence cause A rather than B to be inlined in the FDO on the sampled profile. As A is quite large, inlining it could cause many other functions to fail to get inlined because of the limit on the size of the resulting function. We observed a large degree of differences in inlining decisions of FDO before and after the inconsistency errors are fixed, especially on programs *vortex* and *vpr* (details skipped for lack of space).

It is worth noting that both types of errors have very limited influence on the weighted accuracy of a profile: zero-count errors are on cold blocks, which have small weights in the accuracy calculation; inconsistency errors happen on warm and hot events, but being inconsistency does not prevent those events from being sampled enough times to get a good approximation of their exact values. For unweighted accuracy, zero-count errors play some more substantial role in the calculation, which explains why the unweighted accuracies are much lower than the weighted ones in Section 2. However, the unweighted accuracy still cannot well capture the actual effects of inconsistency errors. These reasons explain why there is no strong correlations between the accuracy of a profile and its usefulness for FDO, despite that profile errors—more specifically, the zero-count and inconsistency errors—affect FDO substantially.

3.2 Simple Profile Rectification

We analyzed the two critical types of sampling errors and showed the performance potential after fixing them with exact profiles. However, in reality we do not have exact profiles in hand when recompiling the programs. We consider two alternative options.

The first is through static analysis. By purely analyzing the program, it tries to find out which basic blocks will be executed for sure and which basic blocks must have the same execution frequency. Recall that the XLC tries to find out straight lines of basic blocks, and only instruments the first basic block in each straight line. This instrumentation optimization is a kind of static analysis. It not only reduces instrumentation overhead, but also maintains the equality relationship among the basic blocks in each straight line. However, to rectify these two types of errors, more sophisticated static analysis is necessary. Furthermore, the conservativeness of static analysis may also form some barriers for the rectification. For two basic blocks that in practice almost always have the same counter values, static analysis cannot give such a conclusion if there is no way to prove that they must have the same counter values. A probabilistic rectification is possible to bring more benefits than conservative static analysis for the nature of program optimizations.

In this work, we choose a second option, which uses a training profile (on a smaller input) to rectify sampled profiles. The rectification is simple. We assign 1 to the counters of the basic blocks, which are covered in the training profile but missed in the sampled profile. There are some other options, to use the exact counter value in the training profile or its scaled version. However, our experiments show that the minimal value change (from 0 to 1) is sufficient. We then identify all consistency groups (e.g., basic blocks that have equal counter values) in the training profile, and maintain the relation in the sampled profile by setting the counter of every block to the average counter value of the consistency set that block belongs to.

We justify this solution by answering three questions in the rest of this section. First, are the basic block coverage pattern and counter equality pattern hold across different ref inputs? Second, is the relatively small training input similar enough with ref input in terms of these two patterns? Third, does the FDO performance from the rectified sampled profiles outperform the performance of just using training profiles?

Pattern Stableness across Ref Inputs. We use a training profile for the rectification based on our claim that once collected, it can be used to rectify many future sampled profiles of production runs. To support this claim, we need to show that the basic block coverage and counter equality patterns are stable across ref inputs. We use 3 ref inputs for each program, and collect their exact profiles. We quantify the basic block coverage pattern stableness by calculating the basic block coverage percentage for each pair of the three exact profiles and get their average. Similarly, to quantify the counter equality pattern stableness, we just replace the basic block coverage percentage with the consistency score.

As Figure 11 (a) shows, the basic block coverage among ref inputs is reasonably stable with a minimum of 70% and an average of 89%. The counter equality pattern similarity is a bit less (on average 85%) due to the difficulty in maintaining it in different profiles.

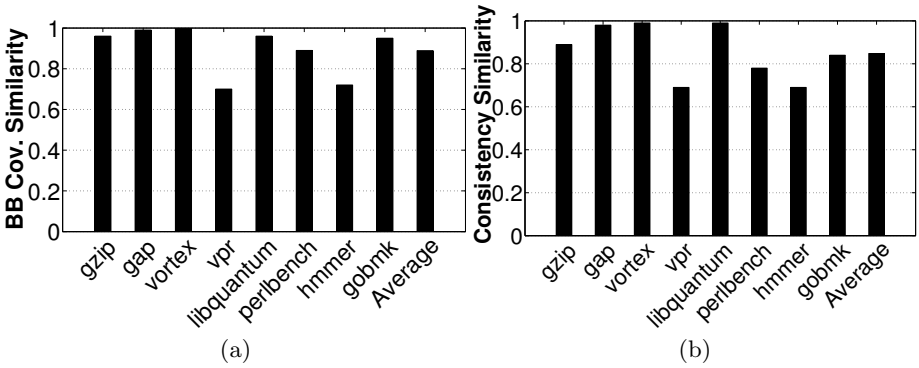


Fig. 11. (a) Basic block coverage pattern similarity across ref inputs. (b) Basic block counter equality pattern similarity across ref inputs.

Pattern Stableness between Train and Ref Inputs. Figure 12 (a) reports the basic block coverage percentages in the training and ref profiles. We observe that on average 87% basic blocks executed on the ref input are also executed on the training input. Figure 12 (b) shows that training profiles' basic block equality patterns are very similar to that of exact profiles, with a consistency score of 83% on average.

The stableness of the patterns across inputs suggests the promise of using training profiles for profile rectification. It is tempting to wonder why not just simply use the training profile as the approximated ref profile for FDO. The reason is that although the training profile carries some value patterns applicable to other profiles, the exact values it contains differ significantly from ref profiles for the high sensitivity of profile values to program inputs. Directly using training profiles for FDO may hence result in some less desirable performance, as we show next.

Performance Results. We use the sampled profiles of the lowest sampling frequencies for evaluation. As Figure 13 shows, the sampling error rectification based on training profiles perform very well by obtaining 92% and 81% of the full FDO benefit from exact profiles for bursty and uniform sampling respectively. Compared to the sampled profiles, the FDO performance benefit recovery from the rectification is 59% and 43% for the two sampling methods. We also include the FDO benefit from purely using training profiles. On average, the rectified sampled profiles of bursty sampling brings 4.6% more speedup than training profiles, which shows the usefulness of training profiles for rectification. For the specially input-sensitive program *perlbench*, the training profile gives even 6%

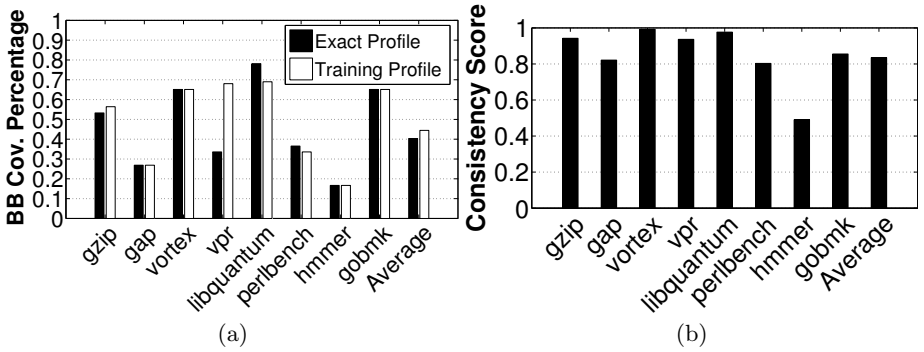


Fig. 12. (a) Basic block coverage pattern similarity between training profiles and exact profiles of ref inputs. (b) Counter equality pattern similarity between training profiles and exact profiles of ref inputs.

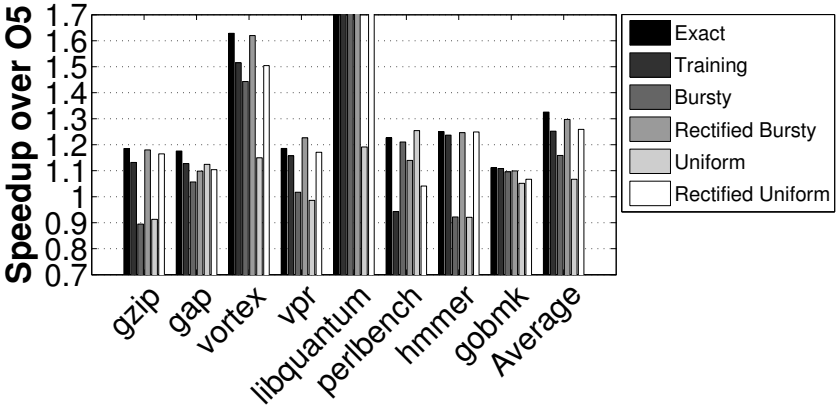


Fig. 13. Speedup comparison of FDO based on exact profiles, training profiles and rectified sampled profiles. (Most bars of "libquantum" are out of the range of the graph; their values are respectively 2.08, 2.09, 1.92, 2, 2.01.)

slowdown, while the sampled profiles—rectified or not—produce 1.04X to 1.25X speedup. It demonstrates that the basic block counter distribution could be very different between training profiles and exact profiles, and so the recompilation based on training profiles may optimize the program in a way not suitable for the ref inputs. We also observe that the rectification reduces FDO benefit by 7% and 21% for bursty and uniform sampling, respectively. This anomaly, along with several other cases in which exact profile does not produce the best performance, implies the imperfect implementation of the FDO due to the complexity in program optimizations, rather than some inherent rules.

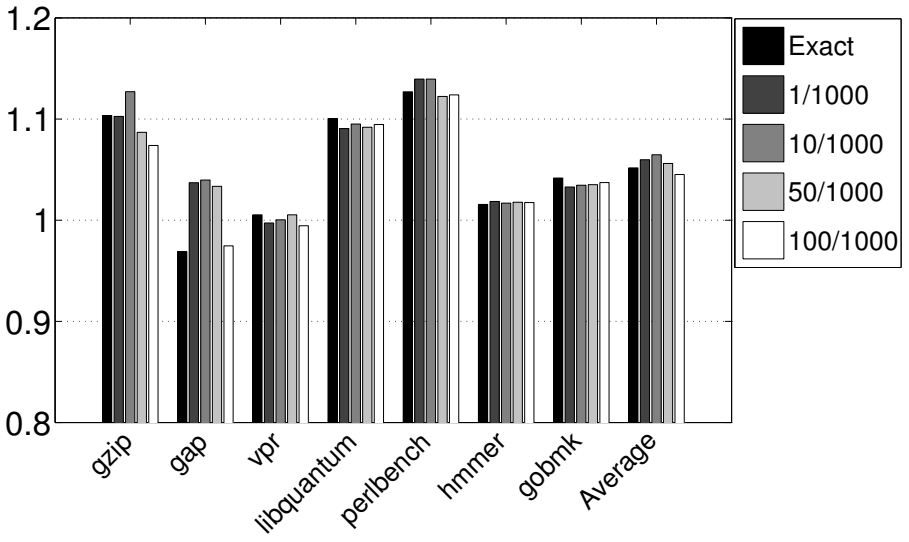


Fig. 14. Speedup by Gcc

3.3 Results from Gcc

Despite the different implementations between Gcc and XLC, most of the insights reported on XLC hold on Gcc. A prominent difference is that Gcc tends to have a smaller degree of speedups by its FDO than those by the FDO of XLC. The reason probably comes from the relatively less sophistication of its FDO implementation.

Figure 14 reports the speedups when bursty sampling of different sampling rates are used. (Vortex is elided as it cannot run through the modified Gcc for some unknown reasons.) The settings include the cases of the highest static compilation, FDO on the exact profiles and fixed sampled profiles. The results show that three benchmarks (*gzip, libquantum, perlbench*) have considerable speedups from FDO when the full profiles are used. For all of them, the rectified profiles help materialize most of the potential of FDO. The four sampling rates, although differing by up to 100 times, do not show much different influence on the FDO benefits when the profile is rectified. On program *gap*, the rectified profiles offers even higher speedups than the full profile. The reason is due to the imperfect design of FDO as mentioned in Section 3.2.

3.4 Discussions

The results in this section indicate that simple profile rectifications go a long way: Despite the simplicity of the profile rectifications through training profiles, the rectified profiles—at even the lowest sampling rate—can help tap into most of the potential of FDO.

Second, when the two kinds of rectifications are applied, speedups replace the performance degradation seen in Section 3.1 when only inconsistency errors

are rectified. It suggests that the two kinds of value patterns have some subtle relations among each other. Fixing them together can help avoid some conflicts subsumed by the relations.

Finally, using a training profile helps explore the potential of profile rectification in this experiment, but after getting the insights that simple rectification to the two types of errors is sufficient, one may choose some other ways to do the rectification. For instance, one could combine sophisticated static program analysis with lightweight profiling on some ambiguous branches to identify the two kinds of value patterns of counters. Combined with cross-production run lightweight profiling [19], the method may provide more seamless integration with the JIT-based runtime engines. Detailed research in this direction is future work.

4 Related Work

Levin and others proposed the use of a Minimum Cost Circulation algorithm to adjust an incomplete edge profile of a control flow graph [14] for post-link optimization. The basic idea is to find minimum adjustment to the edge and basic block weights (i.e., frequencies) in a sampled profile such that after the adjustment, the weights meet the constraints defined by the control flow graph, that is, the sum of the weights of all incoming edges of a block equals the sum of the weights of all outgoing edges of the block. A later study by Chen and others extended the idea to higher level compilation and explored the usage of extra hardware performance counters for alleviating sampling errors [10]. Our work concentrates on the two special types of sampling errors, namely zero-count and inconsistency errors. Both types of errors impose some challenges to the prior approach. For zero-count errors, consider a loop with its preheader block executed once and the loop body executed 1000 times. Due to the inaccuracy in sampling, the sampled frequencies could be 0 for the preheader block, 92 for the loop body and 93 for the loop back edge. The previous algorithm may adjust the back edge to 92 to meet the constraint on incoming and outgoing edges of the loop header block. But that adjustment fails to correct the zero-count error of the preheader block. The algorithm is even less effective in fixing inconsistency errors. The algorithms may be able to adjust the frequencies such that two blocks that are dominator and postdominator of each other have the same frequency. But as Section 2.2 mentions, such kind of relations are already being explored by many compilers by default. Most inconsistency errors we observed happen on blocks across functions or branches. They often reflect dynamic patterns rather than static invariants. Although they are hard to capture by the prior algorithm on static control flows, they are fixable through the statistical rectification method proposed in this work.

In a previous exploration [16], Mytkowicz and others have studied existing profilers and showed that they failed to agree with each other on the identification of hot functions. They found the sources of incorrectness and proposed a prototype of a random sampler to remove the biases in the previous implementations of random samplers. The only study we have found directly on the relation

between profile sampling and the usefulness of profiles for FDO is by Langdale and others [13]. In that study, the authors have used only uniform sampling on machines a decade old. More importantly, the authors used compilers with quite preliminary FDO implementation: The full potential of the FDO on exact profiles is only about 3% speedups. Because of all these limitations, most conclusions from that work are out of date and even contrary to what we observe on modern compilers and machines (e.g., the results on busty sampling.) To the best of our knowledge, the study presented in this current paper is the first systematic study on the relations among sampling, profile accuracy, and profile usefulness *on modern compilers, machines, and sampling methods.* Moreover, we are not aware of previous proposals of the two types of profile error rectification. The novel insights this study obtains are multi-fold on many aspects, including sampling for profiling, FDO, profile rectification, and cross-input stableness of value patterns in profiles.

Many FDO-related studies focus on efficient instrumentation. Knuth and Stevenson show in [12] that they only need to instrument a minimum number of edges of the control flow graph and calculate the counters for all other edges in an offline analysis. Ball and Larus [7] propose an efficient path profiling technique, which encodes each path into a non-negative integer and uses it as an index to update global counters efficiently. In [20], the authors separate interesting paths and profile them with low overhead. Some other studies focus on reducing profiling overhead through sampling. Bond et al. [8] identifies that Ball's path profiling algorithm overhead bottleneck is counter update, and uses sampling to reduce the overhead to provide continuous profiling. Arnold and others [6] reduce instrumentation overhead of a JAVA JIT compilation system by creating a fully instrumented copy for each function and periodically switching execution to that copy to collect profile information.

Many researchers propose to take advantage of different kinds of profiling information for various optimizations. Pettis and Hansen [17] leverage execution counter profile to order procedures and position BBs within each procedure. By exploring the correlations among BBs, this optimization improves greatly code cache performance and reduces branch penalty. Chang and others [9] have implemented an inter-file inliner that automatically uses profile information. Wu [22] explored memory load profiles to find stride patterns and identify the responsible load instructions for prefetching. Rajagopalan and others [18] profile event-based programs to identify commonly occurring event sequences, and reduce the overhead from function indirections. One of our previous papers [21] finds correlations among program behaviors through multiple profiles and applies the technique to runtime version selection.

With the trend of adding more kinds of hardware counters in modern machines, we have seen increasing interests in exploring those hardware resources. Ammons et. al [4] attach hardware counter information on calling context sensitive paths. Adl-Tabatabai and his colleagues [3] leverage hardware counters to smartly inject prefetching instructions in a JIT compilation system.

5 Conclusion

This paper presents a systematic exploration on the relations among sampling rates, profile accuracy, and profile usefulness for FDO. The exploration covers seven factors in four levels. It reveals some counter-intuitive relations, the most prominent of which are that higher sampling rates (within a typical sample rate range) do not lead to more accurate profiles when uniform sampling is used, and more importantly, no matter which sampling method is used, the accuracy of the profiles does not show a strong correlation with their usefulness for FDO. The paper then describes a detailed analysis and points out that two types of sampling errors, *zero-count errors* and *inconsistency errors*, play an essential role in restraining the power of FDO. Based on empirically confirmed cross-input stableness of two kinds of value patterns in profiles, the paper presents a simple way to rectify the two types of errors through statistical patterns. The dramatic enhancement of the FDO benefits concludes that statistical rectification of the two types of errors in a profile is promising in tapping into the full potential of FDO. It also suggests that with profile rectification, sampling rate (and hence sample overhead) can be significantly lowered without hurting the FDO benefits. In addition, the study exposes some subtle relations among the rectification of the two types of errors, and meanwhile, reinforces that bursty sampling is superior to uniform sampling for collecting profiles for FDO.

These multi-fold novel insights provide the first principled understanding in effective collection of profiles for FDO. They may help advance the profile collection in modern runtime systems, and open up many new opportunities for modern program optimizations.

Acknowledgment. We owe the anonymous reviewers of ECOOP'13 our gratitude for their helpful comments to the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0811791 and CAREER Award, DOE Early Career Award, and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, DOE, or IBM.

References

1. Jikes RVM, `http://jikesrvm.org`
2. SPEC CPU benchmarks, `http://www.spec.org/benchmarks.html`
3. Adl-Tabatabai, A., Hudson, R., Serrano, M., Subramoney, S.: Prefetch injection based on hardware monitoring and object metadata. In: PLDI (2004)
4. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. In: PLDI (1997)
5. Arnold, M., Grove, D.: Collecting and exploiting high-accuracy call graph profiles in virtual machines. In: CGO (2005)
6. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: PLDI (2001)

7. Ball, T., Larus, J.R.: Efficient path profiling. In: MICRO (1996)
8. Bond, M.D., McKinley, K.S.: Continuous path and edge profiling. In: MICRO, pp. 130–140 (2005)
9. Chang, P.P., Mahlke, S.A., Chen, W.Y., Hwu, W.: Profile-guided automatic inline expansion for c programs. Software Practice and Experience 22(5) (1992)
10. Chen, D., Vachharajani, N., Hundt, R., Liao, S., Ramasamy, V., Yuan, P., Chen, W., Zheng, W.: Taming hardware event samples for fdo compilation. In: CGO (2010)
11. Hirzel, M., Chilimbi, T.M.: Bursty tracing: A framework for low-overhead temporal profiling. In: Proceedings of ACM Workshop on Feedback-Directed and Dynamic Optimization, Dallas, Texas (2001)
12. Knuth, D., Stevenson, F.: BIT Numerical Mathematics. BIT Numerical Mathematics 13(3), 313–322
13. Langdale, G., Gross, T.: Evaluating the relationship between the usefulness and accuracy of profiles. In: Proc. Workshop on Duplicating, Deconstructing, and Debunking (2003)
14. Levin, R., Haber, G., Newman, I.: Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In: HiPEAC (2008)
15. Mousa, H., Krintz, C.: HPS: Hybrid profiling support. In: PACT (2005)
16. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Evaluating the accuracy of java profilers. In: PLDI (2010)
17. Pettis, K., Hansen, R.C.: Profile guided code positioning. In: PLDI (1990)
18. Rajagopalan, M., Debray, S.K., Hiltunen, M.A., Schlichting, R.D.: Profile-directed optimization of event-based programs. In: PLDI, pp. 106–116 (2002)
19. Tian, K., Zhang, E., Shen, X.: A step towards transparent integration of input-consciousness into dynamic program optimizations. In: OOPSLA (2011)
20. Vaswani, K., Nori, A.V., Chilimbi, T.M.: Preferential path profiling: compactly numbering interesting paths. In: POPL (2007)
21. Wu, B., Zhao, Z., Shen, X., Jiang, Y., Gao, Y., Silvera, R.: Exploiting inter-sequence correlations for program behavior prediction. In: OOPSLA (2012)
22. Wu, Y.: Efficient discovery of regular stride patterns in irregular programs. In: PLDI (2002)

The Shape of Things to Run

Compiling Complex Stream Graphs to Reconfigurable Hardware in Lime

Josh Auerbach, Dave F. Bacon, Perry Cheng, Steve Fink, and Rodric Rabbah

IBM T.J. Watson Research Center, Yorktown Heights, NY 10598 USA

Abstract. Reconfigurable hardware can deliver impressive performance for some applications, when a highly static hardware design closely matches application logic. Obliged to express efficient static hardware structures, hardware designers cannot currently employ abstractions using dynamic features of modern programming languages.

We present the design and implementation of new features in the Lime programming language that admit construction of stream graphs of arbitrary *shape* using the expressive power of an imperative, object-oriented language. The Lime programmer marks computations destined for hardware, and the compiler statically checks these computations for *repeatable structure*. If the check succeeds, the system guarantees it can extract the static structure needed for hardware synthesis.

We describe the language design in detail and present case studies of 10 Lime benchmarks, each successfully synthesized to a Xilinx Virtex 5 FPGA.

1 Introduction

The end of frequency scaling has driven computer architects and developers to parallelism in search of performance improvements. Since multi-core processors can be inefficient and power-hungry, many have turned to specialized accelerators including GPUs and other more radical architectures.

The most radical programmable architecture is reconfigurable hardware in the form of Field-Programmable Gate Arrays (FPGAs). Compiling a program directly into hardware eliminates layers of interpretation, which can dramatically improve performance, power, or energy consumption.

Today, the vast majority of FPGA developers rely exclusively on low-level hardware description languages (HDLs) such as VHDL and Verilog. These HDLs provide low-level abstractions such as bits, arrays of bits, registers, and wires. With low-level abstractions and tools, FPGA development takes much more expertise, time, and effort than software development for comparable functions.

HDL designs derive their efficiency from hardware structures tailored to closely match application logic. The structure of a hardware design represents a dataflow graph, where each node encapsulates some behavior, and the nodes exchange data over wires and queues. An HDL design implements a data flow graph by instantiating hardware modules and explicitly connecting individual wires between modules. These hardware structures must be *static* – the design must

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 679–706, 2013.

fully elaborate all hardware structures at synthesis time, when tools compile an HDL design to a binary circuit representation. Synthesis often takes hours to complete, and may entail exploration of a configuration space of tuning options.

To improve programmer productivity describing data flow graph computations, several software systems provide language support for stream programming. Some streaming systems, such as StreamIt [1] and SPL [2], provide restricted, self-contained languages to describe data flow graphs, so that their structure can be analyzed statically. Other approaches, such as FlumeJava [3], embed operators as first-class objects in a general purpose language, without, however, enabling static extraction of their structure.

Embedding streaming constructs in a general purpose language has many advantages. Specifically, the programmer can use the full power of the language to describe stream graphs, exploiting modern language features and abstractions. For example, modern language features such as higher-order functions and parameterized types allow the developer to encapsulate design patterns in reusable libraries and software components.

Unfortunately, when compiling stream graphs to an FPGA, the power of a general purpose language cuts as a double-edged sword. Modern software patterns tend to abstract and obscure structural information, which must be elaborated statically to synthesize efficient hardware.

A number of previous projects have adopted streaming programming abstractions for reconfigurable hardware [4,5,6,7]. These projects require a separate compile-time language to express stream graphs (often with restricted topologies). We are not aware of any previous work that supports stream graphs as first-class objects in a modern, general-purpose language, and yet still can compile efficient hardware for an FPGA.

This paper describes new features in Lime (a Java-derived language), which marry the benefits of first-class streaming language constructs with the ability to synthesize efficient hardware. In Lime, stream graphs are first-class objects which can be manipulated with the full power of the language.

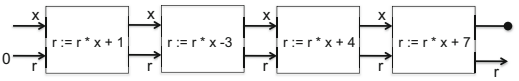
In general, Lime allows the programmer to express graphs whose structure cannot be known until run-time. However, the programmer can denote certain graph expressions for *relocation* to hardware, in which case the language enforces additional invariants using simple local constraints based on compositional language features. When a stream graph construction type-checks as *relocatable*, the language guarantees that the compiler can extract static structure needed to synthesize efficient hardware. Furthermore, we show a language/compiler co-design that allows the system to extract static structure without heroic program analysis and without symbolic execution.

The contributions of this work are:

- *object-oriented language support for first-class stream graphs: tasks and stream graphs are first-class entities in the language, allowing creation of rich structures and abstracting complex topologies into graph creation libraries;*
- *repeatable expressions:* (first introduced in [8]), are generalized and exploited to support extraction of static graph structure;

Inputs:
an integer x
array of coefficients $[a_0, a_1, \ldots, a_n]$
Output:
$f(x)$, $f = a_0x^n + a_1x^{n-1} + \ldots + a_n$
Algorithm:
$r = 0$
for $i \in [0, \ldots, n]$:
$r = r * x + a_i$
return r

a)



b)

Fig. 1. a) Pseudo-code for Horner's rule. b) Pipeline to evaluate $f(x) = x^3 - 3x^2 + 4x + 7$.

- *relocatable task graphs:* a simple syntactic construct to denote stream graphs intended for hardware acceleration. The compiler enforces type-checking constraints (based on repeatability) which guarantee that it can extract the required static graph structure for hardware;
- *implementation:* our compiler implements a limited partial evaluator using Java bytecodes which is sufficient to extract graph structures built using the full feature set of the language (avoiding the need for symbolic execution or aggressive program analysis); and
- *compilation into hardware:* we show that our language is sufficient to express a variety of structured graphs, and additionally can express irreducible graphs and incorporate unstructured imperative code into stream graph construction routines.

2 Overview

Consider a simple motivating example: a stream evaluator for a polynomial. Given a polynomial $f(x)$, when presented a stream of inputs $\{x_0, x_1, \ldots\}$, the program should produce the stream $\{f(x_0), f(x_1), \ldots\}$. Assume a non-functional requirement: we need a pipelined implementation on an FPGA, which consumes and produces one value per cycle.

We base our algorithm on Horner's rule for evaluating a polynomial; see Figure 1a.

Example 1. Consider the polynomial $f(x) = x^3 - 3x^2 + 4x + 7$. Figure 1b shows the structure of a pipelined implementation to evaluate $f(x)$ according to Horner's rule. Each pipeline stage performs one multiply-add, which we assume can synthesize in one cycle. If presented one value x per cycle, this pipeline produces one value $f(x)$ per cycle.

Next, we describe how one could express this stream graph in Lime.

2.1 Stream Graphs in Lime

Lime is based on the Java Programming Language [9], but adds a number of constructs to express invariants helpful when compiling programs to hardware [8].

Here we sketch those Lime constructs needed to understand the motivating example. In Section 3 we describe the relevant features of Lime more completely.

Briefly, the Lime type system includes various types that enforce immutability and restrict side effects. For the purposes of this section, we assert that any method marked with the `local` qualifier has no side effects and can be considered a pure function. Additional, we will use the Lime *tuple* types: the syntax `‘(x,y,z)` indicates a tuple with three elements, and the syntax `‘(int,int,int)` specifies the type meaning "tuples of three integers". Lime supports type inference for local variables; the programmer can elide the type in a local variable declaration, and simply use `var` or `final` instead.

Lime supports a streaming dataflow programming model; a Lime program constructs a *stream graph* by creating *tasks* and composing them into an acyclic graph. A Lime program applies the `task` operator to a "method description" to produce a Lime *task*, a node in a stream dataflow graph.

The full Lime language supports a number of syntactic forms for method descriptions, which correspond to instance methods, and support object state for stateful tasks. For expository purposes, we restrict our attention in this paper to *stateless* Lime tasks constructed from static methods. However, all the concepts presented in this paper translate naturally to the full Lime language, including stateful tasks.

Definition 1 (Task Construction). *Let* `T0 Foo.m(T1, ..., Tk)` *be the signature of a static method* `m` *declared on class* `Foo`*, which takes parameters of types* `T1` *through* `Tk`*, and returns a value of type* `T0`*. If all of the types* `T0 .. Tk` *are* value types *(Section 3.1), then the expression*

```
task Foo.m(T1, ..., Tk)
```

is a task construction *expression.*

If the signature without parameter types `Foo.m` is unambiguous, then `task Foo.m` is accepted as shorthand for the full signature.

A task construction constructs an object of type `Task`, which represents a node in a stream graph. The constructed task takes k inputs, whose types are `T1` through `Tk`. If `T0` is `void`, the task returns zero outputs. If `T0` is a tuple type of cardinality m, the task produces m outputs with types corresponding to the tuple components. Otherwise the task produces one output of type `T0`. Each time it activates, the constructed task consumes inputs, applies the pure function represented by method `m`, and outputs the result.

`Task` is an abstract type – the language provides subclasses of `Task` that describe its shape. For example, `class Filter <IN, OUT> extends Task` is a commonly-used subtype, that describes a task that consumes an input of type `IN` and produces an output of type `OUT`. `Task` and all its subclasses are *value types* (Section 3.1).

The program can eagerly bind (*curry*) one or more input arguments to a task, by specifying the bound values in the task construction expression. The curried expression is evaluated once, at task initialization time.

```
static local `(int, int) ingress(int x) { return `(x, 0); }
static local int egress(int x, int r)   { return r; }
static local `(int, int) update(int x, int r, int coef) { return `(x, r * x + coef); }

var pipe = task ingress => task update(int, int, 1)  => task update(int, int, -3) =>
                           task update(int, int, 4)  => task update(int, int, 7)  =>
                           task egress
```

Fig. 2. Lime code to construct a pipeline to evaluate $f(x) = x^3 - 3x^2 + 4x + 7$

Definition 2 (Task Initializer). *Let* $t =$ `task Foo.m(T1, ..., Tk)` *be a valid task construction. Let* `e` *be a valid expression of type* `Tj` *for* $1 \leq j \leq k$*. Then the expression* t' *which substitutes* `e` *for* `Tj` *in* t*,*

```
task Foo.m(T1, ..., e, ... Tk),
```

is a valid task construction. In t'*,* `e` *is called a* task initializer*. The meaning of* t' *is the same as* t*, where the value of the* j*th parameter is statically bound to the value of* `e`*.*

Lime programs compose tasks into simple stream graphs using the *connect* (`=>`) operator. If t_1 and t_2 are tasks, the expression $t_1 => t_2$ describes a stream graph where the outputs of t_1 flow to the inputs of t_2.

Example 2. Figure 2 shows the Lime code to construct the pipeline for the polynomial $f(x) = x^3 - 3x^2 + 4x + 7$. The resultant pipeline matches the structure described in Figure 1b. Note that each instance of the `update` task statically binds the `coef` input to an appropriate integer coefficient value.

In Figure 2, the structure of the stream graph, meaning its shape and the implementation of each task, is *static* and clearly evident from the code. When compiling to hardware, the compiler must elaborate this structure statically in order to synthesize an efficient hardware design that produces one value per cycle.

2.2 Polynomial Parser

We now turn to a more challenging problem, which motivates the novelties of this paper. Suppose we wish to write a library which can generate circuits for arbitrary polynomials, represented by strings.

In software, general purpose languages naturally support this style of library. Clearly, we can build some abstract data structure that represents a polynomial, and an evaluation engine which interprets the data structure at runtime. We can use the same philosophy to write a library routine that generates Lime stream graphs.

Returning to the example, let's represent a polynomial by an array of `int`, so the polynomial $f(x) = x^3 - 3x^2 + 4x + 7$ corresponds to `int[] f = { 1, -3, 4, 7}`. We need a `parse` method that converts a string representing a polynomial to an array of coefficients, and a method `pipeline` that constructs a Lime task graph from an array of integers. Figure 3a sketches a simple implementation in

```
0  value class string {...}
1  static int[] parse(string s) {...};
2  static Task pipeline(int[] coef) {
3    var pipe = task ingress;
4    for (int c: coef) {
5      pipe = pipe => task update(int,int,c);
6    }
7    return pipe => task egress;
8  }
9  int[] f = parse("x^3 - 3x^2 + 4x + 7");
10 var pipe = pipeline(f);
```

a)

```
0  value class string { ...}
1  static local int[[]] parse(string s) {...};
2  static local Task pipeline(int[[]] coef) {
3    var pipe = task ingress;
4    for (int c: coef) {
5      pipe = pipe => task update(int,int,c);
6    }
7    return pipe => task egress;
8  }
9  final f = parse("x^3 - 3x^2 + 4x + 7");
10 var pipe = ([ pipeline(f) ]);
```

b)

Fig. 3. a) Lime code to generate a pipeline to evaluate a polynomial represented by an array of int. b) Similar code enhanced with new Lime language constructs.

Lime. We elide the details of the `parse` method, which implements basic string processing using imperative operations.

The code in Figure 3a correctly expresses the necessary logic in Lime. However, in contrast to Figure 2, the code in Figure 3a does not mirror the structure of the resultant stream graph for any polynomial. The structure of the stream graph depends on the contents of a string (line 9), relatively complex imperative parsing code (line 1), and a loop (lines 4 - 6) that constructs a task graph.

The Lime runtime system will happily build and interpret this task graph at runtime, running in software. When running in software, the system can construct and interpret fully dynamic graphs, at run-time. However, in order to exploit reconfigurable hardware, the compiler needs more static information. In order to generate hardware efficiently, the compiler needs to determine the structure of the stream graph at *compile-time*.

To determine the structure of a stream graph for the polynomial example, clearly the compiler requires that the string (line 9) which determines the polynomial be known at compile-time. However, even when the string is known, extracting the stream graph structure from vanilla Java code in Figure 3a represents a daunting program analysis challenge. Effectively the compiler must partially evaluate the stream graph constructor for a given input, which carries all the inherent difficulties of binding-time and side-effect analysis for Java.

The main contribution of this paper consists of a language/compiler co-design that makes this problem tractable. We present Lime language extensions to add small but powerful type constraints that allow the compiler to extract the relevant stream graph structure without heroic program analysis. As we shall show, the language remains sufficiently general to express rich structured and unstructured stream graphs.

Figure 3b shows Lime code to construct the pipeline using the new language extensions. The revised code relies on the following Lime concepts:

– *immutable arrays*: The double bracket syntax (`int[[]]`) indicates an array whose contents are immutable. (lines 1, 2)

- *values*: A class marked as a `value` is deeply immutable. Line 0 shows that the library class `lime.lang.string` is a value class. Instances of value classes and immutable arrays are called *values*.
- *local functions*: A method marked as `local` (line 1) cannot write to mutable static data. Thus, a static method must be a pure function if the following conditions hold: i) it is local, ii) all parameters are values, iii) it returns a value.
- *repeatable expressions*: Informally, any expression which is composed from compile-time constants, value constructors, and pure function applications is considered *repeatable*. The compiler can safely evaluate a repeatable expression at compile-time. In the example, the expression `parse("x^3- 3x^2 + 4x + 7")` is repeatable.
- *relocation brackets*: An expression in relocation brackets *e.g.*,`([e])` defines a stream graph, intended to be executed on (*i.e.*, relocated to) a specialized device such as an FPGA. In relocation brackets, `e` must satisfy constraints that guarantee the compiler can extract the relevant static graph structure.

Immutable arrays, values, and local functions were previously presented in [8]. Repeatability in [8] applied only to static fields, and is greatly generalized in this work. Relocation brackets and the analysis to support them is new.

We will fully explain the constraints involving relocation brackets and local methods in this paper. For this example, it suffices to note that if the expression in relocation brackets is repeatable, then it satisfies the constraints. Later we will present other scenarios which relax the restriction to increase expressive power. Note that in Figure 3b, the expression in relocation brackets at line 10 is repeatable (the type `Task` returned by `pipeline` is a value type, as is the argument, and the function is `local`).

What Have We Accomplished? We have introduced language constructs which allow the programmer to write a relatively complex stream graph generating library, using all the imperative facilities of Java. In order to guarantee that the compiler can determine the relevant graph structure statically, we have introduced simple type constraints at the library boundary.

Note that all the type constraints mentioned here are simple local properties which can be checked in a modular fashion. Compile-time evaluation of repeatable expressions would be inter-procedural and arbitrarily complicated if done by conventional means; however, the language design permits a simple *concrete evaluator* to run at compile-time to evaluate repeatable expressions. No complex analysis is required. We will discuss implementation issues in building a robust concrete evaluator in Section 7.

We have shown a simple example to motivate the desire to use arbitrary imperative code to define stream graphs. We have presented only a simple pipeline graph structure; however, the same mechanisms work for all *reducible* [10] graph structures, which include operators to split and join dataflow tokens.

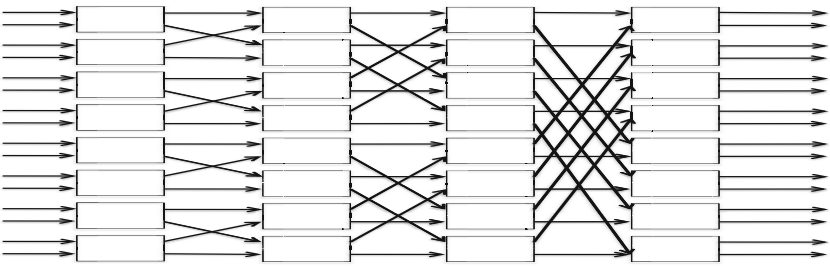


Fig. 4. FFT butterfly network

2.3 Irreducible Graphs

Consider the well-known FFT butterfly stream graph for the decimation-in-time FFT algorithm, shown in Figure 4. This stream graph differs fundamentally from the graphs considered previously in many streaming languages, including previous versions of Lime. The FFT butterfly graph is *irreducible* [10]: it cannot be expressed as a composition of pipelines, splits, and joins.

We have extended Lime with the ability to define stream graphs with *manual connections*. Briefly, we allow the programmer to construct stream graphs with an unstructured graph construction API, which allows arbitrary connections between tasks.

As long as the graph construction code obeys the constraints imposed by relocation brackets, the compiler can extract the relevant structure, even for irreducible graphs created through the programmatic graph API. This property holds even for recursive graph construction routines, which arise frequently in complex graph construction logic.

This facility allows Lime to express a richer graph language than previous streaming languages such as StreamIt[1] which are restricted to reducible graphs. Furthermore, we have implemented complex graph structures such as systolic arrays and FFT, and can statically extract the graph structure and compile the structures to hardware.

3 Lime Preliminaries

Lime is a superset of Java, adding additional language features to express parallelism and locality to exploit heterogeneous architectures. We first review key Lime language features presented previously [8], before introducing the new language contributions in subsequent sections.

3.1 Value Types

Lime introduces a category of *value types* which are immutable (like the primitive types) but are declared similarly to reference types (with fields and methods).

One merely adds the `value` modifier to a type declaration to revise the semantics and obtain additional checking to enforce deep immutability. The fields of a value type are implicitly `final` and must themselves be value types. For example, Lime provides the library type `bit` as a `value enum` with possible values `zero` and `one`. The primitive types inherited from Java are redefined to be value types.

A special array declarator allows some arrays to be values. The type `bit[]` is a mutable array of bits and the type `bit[[]]` is an immutable array of bits (a value type).

The language defines a non-null default type for every non-abstract value type and prohibits null values for such types.

The construction rules for values prohibit cycles, so each value represents a tree that can be linearized and passed-by-value.

3.2 Local Methods

Lime introduces the `local` modifier on methods, which can be used to enforce invariants regarding side effects and isolation. The `local` modifier asserts that a method does not access mutable `static` fields, and only calls other `local` methods. Type checking these rules requires only simple intra-procedural scanning.

The rules give no general guarantee that a `local` method is free of side-effects, since it can modify instance fields in its receiving object or mutable objects reachable from method arguments. However, if a `local` method has only values as arguments and return type, then it is easy to establish that the method is *pure*.

Note that a `local` method established to be pure is *not* obligated to call only pure methods. It is free to call methods that are merely local, since any mutations that may occur inside those methods must be limited to objects created in the activation stack of the outermost `local` method. Any such mutable objects must all die before the outermost method returns, since the outermost pure method must return a value and cannot write to mutable static data structures.

`Local` methods are allowed to read certain static fields (if they are `final` and *repeatable*). Section 5 will present Lime's concept of repeatable expressions.

3.3 Stream Graphs

Section 2 introduced the Lime task constructors and task initializers, which can be used to build stream graph pipelines. Lime also provides a set of *system tasks* called splitters (one input, many outputs) and joiners (many inputs, single output), which can be connected to form a rich set of possible graph structures.

- The *multitask* constructor `task [` $t_1, \ldots, t_k$ `]` constructs a composite task consisting of a vector of k tasks which are *not* connected to each other. Instead, this composite task takes a k-ary tuple as input and produces a k-ary tuple output. The ith component in the input tuple flows to task t_i, which produces the ith component of the output tuple.

```
value class Task {
 Task named(string id);
}
value class TaskGraph extends Task {
 TaskGraph add(Task t);
 TaskGraph connect(Task src, int outPort,
                   Task dest, int inPort);
}
```

```
static local int twice(int x) { return 2*x; }
Task t = task twice;
Task a = t.named("a");
Task b = t.named("b");
TaskGraph tg = new TaskGraph();
tg = tg.add(a);
tg = tg.add(b);
tg = tg.connect(a, 0, b, 0);
```

Fig. 5. a) TaskGraph API b) Example client

- Let `T = ‘(t1, ..., tk)` be a tuple type with cardinality k. Then the constructor `task split T` creates a task that consumes an input of type `T`, and produces k outputs, one for each component of the tuple type. A *splitter* task splits a tuple stream into individual streams for each component.
- Let `T = ‘(t1, ..., tk)` be a tuple type with cardinality k. Then the constructor `task join T` creates a task that consumes k inputs, one for each component of the tuple type, and produces a tuple of type `T`. A *joiner* task creates a stream a tuples from streams of the individual components.

With *connect*, *split*, and *join*, Lime programs can construct any acyclic reducible stream graph shape.

4 Manual Graph Connections

In order to express irreducible graphs such as the FFT butterfly example in Section 2.3, we have extended Lime to support construction of arbitrary stream dataflow graphs.

Figure 5a shows the key API methods for the `TaskGraph` class, which provides a programmatic interface for stream graph construction. The `TaskGraph.add` method adds a task to a graph, and the `connect` method connects an output of one task to the input of another.

Note that `TaskGraph` is a value class – it is deeply immutable. Thus the `add` and `connect` methods create a new `TaskGraph` value, and do not mutate a graph in place. The immutability of tasks plays a key role when reasoning about *repeatability* of task construction code, described in more detail later.

Similarly, the `Task` class itself is also immutable. However, we note that in many complex graphs, such as systolic arrays, the program must build up a network which contains many copies of a particular task. In order to support this, each `Task` instance has a unique string identifier. The method `Task.named(string id)` creates a new copy of a task, but with a different string identifier. The string identifier dictates object identity for task objects, which allows the programmer to distinguish between copies of a functional unit when building complex graphs.

Figure 5b shows Lime code to build a graph equivalent to `task twice => task twice`. Although this simple graph is obviously reducible, it should be clear that a program can use the `TaskGraph` API to build an arbitrary graph structure.

Lime only accepts acyclic graphs. When the manually constructed portion of the graph is further connected using `=>`, the result is checked for acyclicity. In general, this will result in a run-time exception, but if the graph is being relocated, the evaluation technique presented in Section 7 finds the error at compile-time.

5 Repeatability

Next we introduce the concept of *repeatable expressions*, which generalize the repeatable static fields of [8], extending the notion of constancy to arbitrary expressions.

A repeatable expression has no side effects, can be evaluated any number of times, and will always produce the same result. The class of repeatable expressions are those built from repeatable terms composed with pure functions.

Base Terms. The base repeatable terms (those containing no operations) are a superset of the set regarded as "compile-time constant" in Java. First, all literals of value types are repeatable. This includes the primitive type literals defined in Java plus the literals added by Lime for bits, ordinal types (discussed shortly), `value enums`, and string literals.

Generalizing Java's rule for constants, a simple name reference is repeatable if it is a reference to a `final` field or variable that has an explicit repeatable initializer. A qualified name reference (like `Foo.a`) is repeatable if it is a reference to `static final` field that has an explicit repeatable initializer.

Built-in Operators. Lime includes a set of built-in operators which represent pure functions. (*e.g.*,`+, -, *, %, /`). If expression `e` consists of a pure operator applied to arguments that are all repeatable, then `e` is repeatable.

User-Defined Functions and Types. We increase the set of repeatable expressions with two capabilities not supported for constants in Java. First, we exploit the `local` invariants discussed in Section 3.2 to reason about calls to user methods that must be pure functions. So, a method invocation produces a repeatable result if the method is pure and all of its actual arguments are repeatable.

Second, we regard a value creation (with the `new` operator) as repeatable if the constructor is pure and all the actual arguments are repeatable.

Claim: If `e` is a repeatable expression, and evaluation of `e` terminates, then `e` evaluates to the same value in all possible executions. This can be shown by structural induction over the forms of repeatable expressions.

5.1 Ordinal and Bounded Types

When compiling to an FPGA, the generated design must fit in limited physical resources, and cannot exploit a virtual address space. For this reason, the compiler must often be able to compute the size of arrays at compile-time, in order to use scarce logic resources efficiently.

To help the compiler reason about array sizes, Lime includes *bounded array types*. Informally, the type "`int[[N]]`", where `N` is an integer, represents an array of exactly `N` integers. More generally, Lime supports *ordinal types*, where the type "`ordinal N`" [1] represents the set of non-negative integers i where $i < N$. Lime programs can use ordinal types just like any other types in Java; in particular they can be used as type parameters to generic methods. Additionally Lime supports restricted constructs to convert between ordinal types and integer values, so ordinal types represent an extremely limited form of dependent integer types.

The rules for repeatability for integer values extend naturally to define repeatability for ordinal types and type parameters. With repeatable ordinal types and generics, the Lime programmer can build task graphs recursively to express divide-and-conquer algorithms for arrays.

Example 3. Figure 6 sketches a recursive implementation of merge sort, using ordinal type parameters. The type parameter `N` (line 1) indicates the size of the input array. Note that the code constructs a new ordinal type `HALF`, used in the divide and conquer recursion.

Observe that the type system ensures that `sort` is a pure function. Thus, when `sort` is invoked on a repeatable input (line 15), the type system ensures that all type parameters used in the recursive expansion of `sort` are also repeatable. Thus, the compiler can statically determine the bounds of all arrays used in the call to sort at line 15. Furthermore, since `sort` is pure, the result `b` at line 15 is also repeatable and can be computed at compile-time.

Although Figure 6 shows simple single-threaded code, the same concepts apply when constructing stream graphs from generic methods with bounded array inputs and outputs. This pattern arises frequently in stream graphs for Lime programs on FPGAs. Repeatable bounded array types are a key feature in being able to statically bound space usage in hardware designs for complex stream graphs.

5.2 Repeatability Issues

Termination. Lime provides no guarantee that a repeatable expression will terminate without throwing an exception, or even terminate at all. However, the behavior (terminating or not) will be reproducible, and can be monitored at compile-time. When the compiler evaluates repeatable expressions, it checks for exceptions and imposes a time out. Should evaluation not terminate normally in a reasonable interval, the compiler reports the failure as a compile-time error.

Determinism. Invariants for `local` methods and constructors guarantee freedom from side-effects, but not necessarily determinism. We tacitly assume that such methods cannot contain any non-deterministic operations. This assumption is

[1] For expository purposes, we take some liberties to simplify the syntax of ordinal types, as compared to the implemented language design.

```
1  local static <ordinal N> int[[N]] sort(int[[N]] input) {
2     if (N.size > 1) {
3        final int HALF = N.size/2;
4        int[[HALF]] low = lowerHalf(input);
5        int[[HALF]] high = upperHalf(input);
6        low = MergeSort.<HALF>sort(low);
7        high = MergeSort.<HALF>sort(high);
8        return merge(low,high);
9      } else {
10        return input;
11     }
12 }
13 local static <ordinal N, ordinal M> int[[M]] merge(int[[N]] a, int[[N]] b) { ... }
14 final int[[8]] a = {4, 6, 2, 8, 9, 4, 3, 12 };
15 final b = sort(a);
```

Fig. 6. Merge Sort

true today in Lime, because Lime has no core language constructs that are non-deterministic. The type system prevents local methods from calling native code or across a foreign function interface.

Generalizing Repeatability. The current definition of repeatability includes two pragmatic compromises. First, we insist (as with Java compile-time constants) that any `final` fields must first have an *explicit* initializer before we consider whether that initializer is repeatable. Second, we limit qualified names like `Foo.a` to the case where they denote a static field. While it would be possible to relax both restrictions, it would complicate the engineering of the repeatable expression evaluator, and also involve more complex rules for the user to understand. We elide the technical details due to space constraints. Section 7 discusses implementation issues in greater detail.

6 Relocation Expressions

In this Section, we introduce the language constructs that guarantee that the compiler can extract static information about stream graphs, needed in order to relocate a stream graph computation from software onto an FPGA.

Relocation Expressions. If `e` is a Lime expression, we introduce the syntax `([e])`, which we call a *relocation expression* , using "relocation brackets" syntax.

A relocation expression type-checks if either a) `e` is repeatable, or b) `e` satisfies additional constraints specific to unrepeatable task initializers (Definition 2) for stream graphs. We next discuss each case and explain the additional constraints which define case b).

The key insight is that if `e` is a relocation expression which generates a stream graph, then the compiler guarantees that it can extract static structural information (hereafter called *stream graph structure*) sufficient to enable hardware synthesis.

Definition 3 (Stream Graph Structure). *Let g be an object of type* `Task` *(*i.e.*, a stream graph). A stream graph's* structure *consists of* [2]

1. *the topology of the graph, a canonical form of its nodes and connections,*
2. *for each edge, the type of all values that flow on it,*
3. *for each node, the Lime method providing its behavior, and*
4. *for each method parameter of the previous, whether or not it is curried (*i.e.*, constructed via a task initializer as per Definition 2).*

For part 4, the actual value bound to the parameter is not considered part of graph structure. Therefore, the stream graph structure does not completely determine the function to be executed in each node. The "code" is determined by part 3, the curried signature is refined by part 4, but the task initializers (see Definition 2) are still unknown.

6.1 Repeatable Stream Graph Expressions

Clearly, when a stream graph expression is repeatable, the compiler can fully evaluate the expression, and walk the resultant data structure to determine the graph structure. In Section 7 we will discuss implementation details relating to the compile-time repeatable graph evaluator. We have already seen an example of a repeatable graph, in the polynomial example of Section 2.

Stream graph repeatability motivates the design decision discussed in Section 4, where `Task`s are designed as immutable values, allowing us to reason about repeatability for library methods that produce and consume `Task` objects.

As a simple example, consider:

```
static local Task connect(Filter<int,int> a, Filter<int,int> b) { return a => b; }
```

If we want to use `connect` in a repeatable (or relocatable) expression, then the type system must establish that `connect` is a pure function. Recall that we have this guarantee for local methods that produce and consume *values*. Thus Lime Task objects are deeply immutable, as discussed in Section 4.

6.2 Unrepeatable Task Initializations

In Section 2, we considered a pipelined implementation for polynomial evaluation, where the degree and coefficients to the polynomial were static. In this case, both the stream graph structure and all the task initializers are repeatable, so all the node functions were completely determined.

However, we can also efficiently support hardware stream graphs where the stream graph structure is repeatable, but the *functions* of individual nodes in the graph depend on dynamic data provided through unrepeatable task initializers. We now present language extensions to support this.

Consider a variant of the polynomial pipeline for functions of the form $f(x) = a_0x^3 + a_1x^2 + a_2x + a_3$, where the degree of the polynomial is fixed at 3, but the coefficients of the polynomial are unknown at compile-time.

[2] We elide details specific to stateful tasks in the full Lime language.

```
int[4] a = readFromInput();     // assumed dynamic
var pipe = ([ task ingress => task update(int, int, a[0])  =>
                              task update(int, int, a[1]) =>
                              task update(int, int, a[2])  =>
                              task update(int, int, a[3])  =>
                              task egress])
```

Fig. 7. Lime code to construct a pipeline to evaluate a 3rd-degree polynomial $f(x) = a_0x^3 + a_1x^2 + a_2x + a_3$, where the coefficients a_i are not repeatable. Refer to Figure 2 for definitions of `update`, `ingress`, and `egress`.

Intraprocedural Case. Figure 7 shows Lime code to build a stream graph for this problem. In this case, we assume that the coefficient array `a` is *not* repeatable.

Clearly the relocated expression in Figure 7 is not repeatable. To allow this expression to type-check, we relax the type-checking rules.

For the moment, consider the subset of Lime which excludes procedure calls.

Definition 4 (Relocatable Expressions (no calls)). *A (legal) Lime expression* `e` *is* relocatable *if and only if one of the following holds:*

1. `e` *is repeatable*
2. `e` *is of the form* `task M.foo(p1, ... pk)`,
3. `e` *is of the form* `e1 => e2` *where both* `e1` *and* `e2` *are relocatable*
4. `e` *is of the form* `task [e1, ..., ek]` *where each expression* $\texttt{e}_i$ *is relocatable,*
5. `e` *is of the form* `split e1` *or* `join e1`*, where* `e1` *is relocatable*
6. `e` *is of the form* `e1.add(e2)` *where both* `e1` *and* `e2` *are relocatable*
7. `e` *is of the form* `e1.connect(e2,e3)` *where* `e1`*,* `e2` *and* `e3` *are relocatable*

Case 2 represents the interesting case — it allows a relocatable expression to use unrepeatable expressions as *task initializers* (recall Definition 2). This case allows the stream graph in Figure 7 to type check as relocatable. Specifically, the unrepeatable coefficients `a[i]` appear only inside expressions of the form `task e`.

Intuitively, this definition of relocatable constrains the code such that the stream graph structure is repeatable, but the logic that implements each user task in the graph can use runtime values. The system can implement this pattern efficiently in hardware by laying out the stream graph statically, and laying down wires to route the dynamic values to the appropriate functions at runtime.

Interprocedural Case. Next, we extend the definition of relocatable expressions to support procedure calls, so we can encapsulate stream graph constructions in a library, even when they employ unrepeatable values as task initializers.

Figure 8 shows the library method encapsulation for the running example. Note that the expression in relocation brackets at line 10 now contains a procedure call.

```
1    static local Filter<int,int> thirdDegree(task int a0, task int a1,
2                                             task int a2, task int a3) {
3       return task ingress => task update(int, int, a0)  =>
4                              task update(int, int, a1) =>
5                              task update(int, int, a2)  =>
6                              task update(int, int, a3)  =>
7                              task egress;
8    }
9    int[4] a = readFromInput();     // assumed dynamic
10   var pipe = ([ thirdDegree(a[0], a[1], a[2], a[3]) ]);
```

Fig. 8. Lime code to construct a pipeline to evaluate a 3rd-degree polynomial $f(x) = a_0x^3 + a_1x^2 + a_2x + a_3$, where the coefficients a_i are not repeatable. Refer to Figure 2 for definitions of `update`, `ingress`, and `egress`.

We add additional type qualifiers that pass constraints about relocatable expressions across procedure boundaries, but still allow modular local type checking. We re-use the `task` keyword for this purpose – we now allow the `task` keyword as a type qualifier on formal parameters (lines 1 and 2 in the figure).

When a `task` qualifier decorates a formal parameter p of a method m, we call p a *dynamic parameter*. There exist only two legal ways p can appear in expressions inside m:

1. In an expression `task M.foo(p1, ... pk)`, a dynamic parameter p can appear as a bound value for a task initializer `pj`.
2. p can be used as the actual parameter in a call, where the corresponding formal parameter q in the callee is a dynamic parameter.

Any other use of p fails to type check.

In Figure 8, note that each formal parameter of `thirdDegree` is dynamic, but the method type checks because all uses of formal parameters in the procedure satisfy condition 1.

Definition 5 (Relocatable procedure calls). *A procedure call expression* `M.foo(p1, ..., pk)` *is* relocatable *if and only if for each* $j, 1 \leq j \leq k$, *either*

- `pj` *is repeatable, or*
- *the* j*th formal parameter of* `M.foo` *is a dynamic parameter.*

Borrowing a concept from the partial evaluation literature [11], we define an expression `e` to be *oblivious* if, during the evaluation of `e`, every conditional expression evaluated is *repeatable*. Intuitively, if an expression is oblivious, then its evaluation will follow the same control flow branches in every possible environment.

Claim: All Relocatable Expressions Are Oblivious. This property is simple to establish with structural induction on the shape of relocatable expressions. If a relocatable expression is repeatable, obviously it is oblivious. Otherwise, it suffices to note that in each syntactic form listed in definitions 4 and 5, no unrepeatable values can affect control flow.

We next establish a key property, that allows the compiler to extract the shape of relocatable expressions without aggressive program analysis.

Repeatable Structure Property. Suppose a call expression `e = M.foo(e1, ... p, .. ek)` is relocatable, where all actual parameters except `p` are repeatable, and `p` corresponds to a dynamic parameter of `M.foo` of type T. Assume the evaluation of `e` terminates without an exception, producing the stream graph object g_1. Now let `e2 = M.foo(e1, ..., p2, ... ek)` be the expression `e`, substituting any value `p2` of type T for `p`. Then evaluating `e2` terminates without an exception, producing a stream graph object g_2. Furthermore, g_1 and g_2 have the same stream graph structure.

Informally, we can establish this property with an argument based on information flow [12]. We consider the body of `M.foo` as a function with k inputs, where the ith input in_i is the dynamic parameter corresponding to actual parameter `p`. We note that the type checking rules for dynamic parameters guarantee that no statement in `M.foo` can be control-dependent on in_i, and only task constructions can be data dependent on in_i. Thus the effects of in_i on the object resulting from evaluating either `e1` or `e2` must be confined to task constructions. Thus the stream graph structure must be repeatable.

This key property allows our system to evaluate relocatable expressions that produce stream graphs at compile-time, substituting place-holders during evaluation for any dynamic parameters. The claim shows that the structure of the resultant stream graph does not depend on dynamic parameters – instead, dynamic parameters may only flow untouched to task initializers. With this property, the compiler can establish the stream graph structure for relocatable graphs with a relatively simple concrete evaluator, which will be described in Section 7.

7 Implementation

This section addresses implementation issues regarding design of compiler support to extract graph structure. We first discuss issues with repeatable expressions, and then consider partial evaluation for relocatable expressions with unrepeatable sub-expressions.

In a functional language which represents programs as values, a repeatable expression evaluator would be trivial (*e.g.*, `eval e` in Lisp). However, Lime, an imperative language based on Java, does not represent programs as values. Like a Java compiler, the Lime compiler generates JVM bytecodes. So, the compiler can employ the JVM to evaluate repeatable expressions at compile-time.

Namely, our compiler generates bytecode representations of repeatable expressions called *snippets*. The *snippet evaluator* implementation raises some design questions:

1. How do we manage the Java virtual machine runtime environment when running the snippet evaluator?
2. How can the result of a snippet evaluation (an Object or primitive value) be translated to a useful compile-time representation?

3. Relocatable expressions can include unrepeatable dynamic parameters, and these can't be evaluated at compile time, so how does the snippet evaluator perform the implied *partial evaluation* task?

7.1 Runtime Environment for Snippet Evaluation

The Lime compiler generates bytecode representations of all user code *before* running any snippet evaluation. So the snippet evaluator can run with a JVM classpath that includes all the generated bytecode. This classpath reflects the anticipated runtime environment at the granularity of packages and visible classes.

In order to reproduce within-class scoping of names that appear in the expression, the compiler constructs a *snippet method* which represents a repeatable expression. A snippet method has no parameters, and is declared in the class in which the repeatable expression occurs. To build a snippet method, the compiler first creates a single return statement with a copy of the expression. That is, if the expression is `i + j`, then the snippet method starts out as

```
private static int snippetMethod12345() { return i + j; }
```

This method will not type-resolve, since `i` and `j` are variables with arbitrary bindings. The second step visits all the names in the expression and either determines that the correct `i` and/or `j` will actually be in scope, or *replays* the declaration(s) of `i` or `j` inside the method. Since all names are resolved at this point, this analysis can be done accurately.

From the definition of repeatability, we know that any qualified names (e.g. `b.i`) denote `static` fields. If a name refers to a static repeatable field, then no additional steps are required, since the scope already binds the name. If a simple name refers to a local variable, the compiler replays the variable declaration inside the snippet method. If a simple name refers to an instance field defined in the encompassing class or one of its supertypes, then the compiler generates an equivalent local variable declaration in place of the field declaration. When a variable or instance field declaration is replayed, it might trigger transitive replay of other variables or instance fields referenced in the declaration.

The replay strategy is sound for the following reasons. If the snippet is based on a fully repeatable expression, `i` and `j` must denote `final` variables or fields with explicit repeatable initializations. If the expression contains unrepeatable task initializers, they are replaced by placeholders (which have repeatable behavior and don't include name references).

We can now see why supporting non-static qualified names complicates the analysis (see Section 5.2). It is far more difficult to replay the sequence of declarations backing such names since some segments represent objects whose creations have already occurred while others are just field references. In general, one might not even have the source for the class that defines the type of the object or in which the object was created.

7.2 Interpreting Snippet Evaluation Results

Translating a runtime value back into a compile-time representation is simplified by the fact that repeatable expressions always produce Lime values, which are containment trees with no cycles or internal aliases. Translation from runtime back to a compile-time representation can use any of the following techniques.

- **Literal:** If the value is of a type that has a literal representation, use the literal.
- **Default:** If the value corresponds to the default value of its type, use a standard Lime expression to produce that default value.
- **Reconstruction:** Inspect the value's structure (which is alias-free and acyclic) and build a compile-time representation of that structure for use by later compiler phases.
- Otherwise, if the value is an array and its elements can be represented by the previous rules, construct the appropriate array literal.

The present implementation uses the reconstruction technique for the stream graph structure and uses the literal or default technique or their array generalizations for any repeatable task initializers it finds. Unrepeatable task initializers (and those repeatable ones that can only be encoded by reconstruction) are handled using the technique of the next section. In the future we expect to handle more repeatable task initializers by reconstruction, yielding more efficient code.

7.3 Partial Evaluation

Section 6 defined relocatable expressions so that the stream graph structure was required to be repeatable but task initializers could be unrepeatable. The Repeatable Structure Property gives the key insight that allows us to use the snippet evaluator for all relocatable expressions, even with some unrepeatable parameters: we know that we can substitute *any* legal value of the correct type for a dynamic parameter, and the resulting objects from evaluation will have the same stream graph structure. So, in snippet evaluation, we simply generate a unique placeholder value for each unrepeatable parameter, and run the snippet with an unmodified JVM. In the resulting object, the placeholders may flow to task initialization parameters, but (from the Repeatable Structure Property) cannot affect any other aspect of the computation.

As a result, the evaluated object must have the same stream graph structure as the stream graph that will arise at runtime. The implementation may or may not choose to evaluate repeatable parameters that flow to task initializers. Dynamic parameters will be clearly identified by placeholder values in the resultant stream graph object. When interpreting the resultant stream graph object, the compiler maps placeholder values to the appropriate expressions in the generated code, which causes dynamic parameters to flow to generated tasks at runtime.

Table 1. Benchmarks and Results

Benchmark	Description	Graph Size	Param. Value	Static `task`	Dynamic `task`	Idiom	Freq. (MHz)
Beamformer	directional audio signal processing	$9 + 5(C + B)$	$C = 12$ $B = 4$	19	89	map	113
Channel vocoder	voice coder	$9 + 6(N - 1)$	$N = 17$	15	105	repeat	78-133
DES	encryption	$6 + N$	$N = 16$	7	22	recursive	148
FFT	FFT, bulk pipeline	$6log_2(N)$	$N = 32$	9	24	recursive	130
Filterbank	signal processing	$4 + 10N$	$N = 8$	14	84	map	119-161
FM radio	signal processing	$8 + 9N$	$N = 10$	17	98	map	135-312
iDCT	2D inverse DCT	$5 + 3N$	$N = 8$	8	29	repeat	144
Mergesort	sort	$\frac{9N}{2} - 6$	$N = 16$	9	66	d&c	119
Vocoder	rate reducing coder	$51 + 7N$	$N = 15$	58	156	map	181-236
FFT butterfly	FFT butterfly	$2 + \frac{3Nlog_2(N)}{2}$	$N = 8$	5	38	manual	134

8 Results and Evaluation

To evaluate the features described in this paper, we implemented a number of benchmarks, listed in Table 1. All but the irreducible benchmark (`FFT butterfly`) are drawn from the StreamIt benchmark suite [13]. We chose these as a representative sample of reducible graphs, constructed using iterative and recursive coding styles.

Our goal was to evaluate Lime support for stream graph construction using rich abstractions and control flow constructs that include conditionals, loops, and recursion. In the following, we highlight the following findings:

1. Lime permits compact graph construction code using a number of idioms which may be factored into library methods.
2. Graph construction may be parameterized in terms of size via repeatable parameters, and function via tasks as first-class values.
3. The Lime compiler succeeds in extracting the task graphs and synthesizes the Lime code into FPGA circuits.

8.1 Benchmark Characteristics

Each of the streaming applications considered takes a parameter which indicates a problem size. A particular problem size induces the shape of a task graph, which dictates the level of task-parallelism inherent to the algorithm. Table 1 includes the formula for calculating the graph size for each of the benchmarks. In all but one case, a single parameter N defines the problem size. The exception is `beamformer` which is parameterized by two values. The table includes the actual parameter values used for graph extraction and FPGA synthesis.

For each benchmark, we inspected the Lime code and counted the static occurrences of the `task` operator. These are labeled *static tasks* in the table and provide an indication of code complexity. In contrast, the *dynamic tasks* is the size of the graph after graph extraction. It is computed by evaluating the size formula for the given parameter values.

```
task Filter<bit[[64]], bit[[64]]> recursive(task KeySchedule keys, int round) {
   if (round == 0)
      return task des.F(keys.lookup(round), true, bit[[64]]);
   else return recursive(keys, round - 1) =>
               task des.F(keys.lookup(round), round != 15, bit[[64]]);
}

local Filter<bit[[64]], bit[[64]]>[[]] makeFilters(KeySchedule keys) {
    final coder = new Filter<bit[[64]], bit[[64]]>[16];
    for (int round = 0; round < 16; round++)
       coder[round] = task des.F(keys.lookup(round), round != 15, bit[[64]]);
    return new Filter<bit[[64]], bit[[64]]>[[]](coder);
}

public class Idioms {
   static task <V extends Value> Filter<V,V> pipeline(Filter<V,V>[[]] filters) {
      var pipe = filters[0];
      for (int i = 1; i < filters.length; i++)
         pipe = pipe => filters[i];
      return pipe;
   }
}
```

Fig. 9. Recursive graph construction for `DES`

The benchmarks considered exhibit five dominant coding patterns, shown in the column labeled *idioms*:

- **Recursive:** Constructs a graph using recursion. The idiom is useful for constructing a sequence of connected tasks.
- **Divide and Conquer:** A form of recursive graph construction for divide and conquer (d&c) algorithms.
- **Map:** Constructs a $k - ary$ multitask from a single task, and curries the task's vector-position into the task worker method. The map is useful for data- and task-parallel multitasks that operate on partitioned streams.
- **Repeat:** Constructs a $k - ary$ multitask from one or more tasks, currying the task's vector-position into the task worker method. Unlike the map idiom where the tasks operate on a partitioned stream, here every task operates on identical values.
- **Manual:** Constructs arbitrary acyclic graphs using the manual task and connect API. The idiom is most useful for irreducible graphs (*e.g.*, butterfly) but is applicable for reducible topologies as well (*e.g.*, reduction tree).

The ability to treat stream graphs and tasks as first-class objects makes it possible to factor the graph construction idioms into library utility methods. We illustrate a few such examples in the sections that follow.

8.2 Recursive

Three of the benchmarks build graphs using a recursive idiom. Figure 9 shows an example that builds a sequence of tasks that perform the encryption required by `DES`. In this sequence, all but the last task behave identically modulo the curried encryption key (a `KeySchedule`). The last stage performs a bit reversal, indicated by the curried task initializer expression `round != 15`.

```
public class Idioms {
   static task <V extends Value, ordinal N> Task dnc(Task t) {
      final HALF = N.size/2;
      if (N.size > 2)
         return (V #) =>
                task split V[[2]] =>
                task [ Idioms.<V, HALF>dnc(t), Idioms.<V, HALF>dnc(t) ] =>
                task join V[[2]] =>
                (# V[[N]]) => t => (# V);
       else return t;
    }
}
```

Fig. 10. Graph construction using divide and conquer

At each level of the recursion, the graph grows by one task. Although it is convenient to express graph construction in this way, it is often easier to create an array of filters and chain them together using a common utility method. We illustrate this in Figure 9: `makeFilters` creates the array and the library method `Idioms.pipeline` constructs the pipeline.

The graph extracted from `([ recursive(new KeySchedule(), 15) ])` is structurally equal to that constructed with the following expression:

```
([ Idioms.<bit[[64]]>pipeline(makeFilters(new KeySchedule())) ]).
```

8.3 Divide and Conquer

Divide and conquer extends the recursive idiom with parameterized ordinal types (Section 5.1). We present an example of a library utility which exploits first-class task values with higher-order logic.

Figure 10 shows a generic method `Idioms.dnc` parameterized by the type `V` of the values flowing between tasks and the input size `N`. This method builds a graph that divides the input recursively until the base case is reached, connects a task `t` to perform the desired computation, and inserts joiners to combine the results from each level of recursion.

The `mergesort` benchmark uses this idiom to construct its task graph as in `Idioms.<int, 16>dnc(task Merge.sort)` where `task Merge.sort` creates a task to sort a given (merged) array of integers.

The example relies on Lime *matchers* which appear as `#` in the code. This paper did not present matchers, but they were explained in previous work [8]. Conceptually, the simplest matchers (as in this example) perform aggregation to convert a stream of `V` to `V[[N]]` or deaggregation for the reverse conversion. Lime provides type inference across the connect operator so that the left or right side of the conversion may be omitted.

8.4 Map and Repeat

Iterative and recursive construction predominantly serve to construct sequences of connected tasks. An alternate idiom uses multitask constructors (Section 3.3), which construct vectors of tasks not directly connected to each other.

```
public class Beamformer {
   static task Task makeBeams(int N) {
      var beams = new Task[N];
      for (int b = 0; b < N; b++) beams[b] = makeBeam(b);
      return task [ beams ];
   }

   static local Task makeBeam(int id) { return task Beamformer.formBeam(id, float); }
   static local float formBeam(int id, float val) { ... }
}

public class Idioms {
    static task <ordinal N, V extends Value> Task repeat(Filter<V,V>[[N]] filters) {
        return (V # V[[N]], repeat N.size) =>
               task split V[[N]] => task [ filters ] => task join V[[N]];
    }
}
```

Fig. 11. Multitask construction example for map and repeat idioms

Figure 11 shows an example, drawn from the `beamformer` benchmark and simplified for exposition. The `makeBeams` method initializes an array of tasks and then returns the multitask composition of the array elements. The Lime *map* operator (`@`) [8] permits a more concise encoding as

```
static task Task makeBeams(int N) {
   return task [ @ makeBeam(indices(N)) ]; }
```

but we omit the details of the map and `indices` method for lack of space.

We distinguish between two classes of multitasks: those that operate on partitioned streams and constructed using map, and those that operate on a repeated stream. In the former, a single stream is split and distributed to each of the tasks. In the latter, the values in a stream are repeated k times immediately before the splitter. The end result is that each of the connected tasks observe and operate on the same values.

We implemented a library utility method called `Idioms.repeat` that accepts a multitask, and returns a graph consisting of a task that repeats values the required number of times, and connects it to a splitter, multitask and joiner. The example (Figure 11) illustrates another feature of matchers (*i.e.*, the repeat count) that is not covered in this paper. Conceptually, the matcher repeats every value it consumes `N` times on every invocation.

8.5 Manual

The preceding examples all exhibit reducible graph topologies. Other important topologies can only be expressed with manual connections. These include not only irreducible graphs, but also use cases that are simply easier to express using a richer programmatic interface.

Toward this end, the Lime manual connection API (Figure 5) has proved useful. Using manual connection, we created several library utilities to express a variety of graph shapes. These include butterfly networks, systolic arrays and reduction trees. In each case, the library utilities establish the desired shape, parameterized by a small number of values, and allow the programmer to pass in tasks as first-class values to connect internally.

We rely on the power of Lime to encapsulate complex manual graph construction algorithms in libraries using higher-order functions, while still enjoying parameterized types and sizes, type safety, and repeatable graph shapes. The ability to write the graph construction code in the same language and semantic domain as the rest of the application also means a single development and debugging environment is needed. This is especially helpful for the construction of irreducible graphs where the code is relatively more complex compared to idioms illustrated earlier.

8.6 Extraction and Synthesis Results

The Lime compiler successfully extracted the relocatable graphs in each of the benchmarks. An FPGA-specific backend compiled the Lime code to Verilog and using a commercial toolchain, synthesized the code into circuits suitable for programming an FPGA. We used Xilinx ISE version 14.1 to synthesize designs for a Virtex 5 XCV5LX330T FPGA. The last column of Table 1 reports the clock frequencies achieved for each of the benchmarks.

All of the benchmarks are written so there is one extractable graph; however the FPGA backend cannot generate code for every task in some of the graphs. Usually limitations arise from matcher tasks with advanced features not discussed in this paper. In such cases, the compiler partitions the task graph into the largest non-overlapping subgraphs, and synthesizes each partition independently. For these, we report the range of frequencies achieved for the subgraphs. We are working to address the implementation gaps in our FPGA compiler so that the entire extracted graph is compiled into one large circuit.

The synthesis results provide evidence that the graph extraction and relocation concepts are sufficiently powerful to synthesize task graphs into FPGA circuits from high-level object-oriented code. The compiler can fully pipeline the logic for task graphs that consist only of matchers and tasks created from pure methods that are considered combinational. The polynomial evaluation algorithm illustrated throughout the paper is one such example. The data encryption standard `des` is another. The remaining benchmarks contain non-combinatorial tasks that our compiler does not yet fully pipeline. Hence, the achieved frequency is not an absolute measure of throughput for the majority of the benchmarks.

9 Related Work

Streams-C [5] pioneered the idea of using streaming language extensions for programming FPGAs. Streams-C provided a simple, static macro language for describing stream graphs, and let the programmer express kernel logic in a subset of C. The stream graph macro language, following the Communicating Sequential Processes (CSP) model [14], allowed the construction of arbitrary graph shapes. The graph language did not include any substantial constructs to support abstraction or code reuse.

Several other projects have targeted FPGAs with CSP-style streaming constructs and kernels written in C subsets. For example, ASC [6] incorporates streaming directives into C kernels based on loops. Plavec *et al.*, [4] marry kernels in C with the Brook [15] streaming language. These projects did not address issues relating to abstraction, encapsulation, or reuse of stream graph construction algorithms.

Lime support for streaming was inspired by the influential StreamIt language [16]. In particular, Lime builds on work by Hormati *et al.*, [7] which compiles StreamIt to FPGAs. Unlike work based on CSP, StreamIt provides a structured language suitable for reducible stream graphs. As in Lime, expressions which describe stream graphs may be parameterized by compile-time constants. As we have shown, Lime supports a much more general notion of compile-time constants than previous object-oriented languages.

StreamIt provides a small language for specifying stream graphs; in contrast, Lime embeds the stream graph construction in a general purpose object-oriented language. As such, Lime provides a richer language with more advanced abstraction, which can be used to construct stream graphs. We have shown innovations in Lime which allow it to support irreducible stream graphs constructed with complex code, which can still be synthesized to an FPGA.

Like StreamIt, SPL [2] provides a restricted separate language for describing stream graphs. Recently, Hirzel and Gedik [2] introduced higher-order composite operators into SPL, to support richer abstractions for describing stream graphs. Hirzel and Gedik introduce specialized macros to express higher-order constructs. In contrast, our approach inherits higher-order constructs for free by embedding graphs in a general purpose object-oriented language.

The Flask project [17] presents a domain-specific streaming language embedded in Haskell, targeting sensor networks. Flask guarantees static structure of stream graphs by staging the static graph construction code earlier than the dynamic object programs which run on devices, using concepts introduced for two-level meta-programming in functional languages [18]. The Lime constructs presented here might be considered an approach to support limited multistage programming in a Java-like language.

9.1 Partial Evaluation

Our work is related to strategies for off-line partial evaluation [11]. In particular, Lime constructs presented here can be viewed as binding-time annotations, which specify a division to drive off-line partial evaluation of certain Lime constructs.

Several previous systems introduced annotations into programming languages to drive specialization. Notable systems have targeted ML [19] and the C programming language [20,21,22]. Our work, targeting an imperative language, has several aspects in common with previous systems for C.

Consel and Noel presented Tempo, a system for runtime specialization of C programs [22]. Tempo adds annotations to C programs to mark variables as static or dynamic, including constructs to denote *holes* to represent runtime values that should be plugged in to specialized template code at runtime. Tempo

incorporates non-trivial alias analysis in order to propagate annotations during binding-time analysis. Tempo supports more than oblivious algorithms, and includes significant support for expanding specialization templates at runtime.

Poletto *et al.* presented 'C [20] in which the programmer could mark general expressions as "runtime constants", and the system would defer code generation using such values until runtime. In DyC [21], the programmer could annotate function parameters to provide an initial division of the program into static and dynamic components. The DyC system provided significant runtime support to specialize the dynamic division online, including support for polyvariant specialization and division, and interprocedural specialization at runtime.

Similar to these systems, Lime allows the programmer to specify an initial division which drives partial evaluation. However, Lime differs from the previous work for C in several significant aspects. First, Lime supports solely off-line specialization, whereas many previous systems such as 'C, DyC, and Fabius [19] provide significant support for runtime specialization. Lime targets code generation for devices such as FPGAs, where runtime specialization is not practical with current synthesis technology. Instead, Lime restricts the relevant code to oblivious algorithms amenable to off-line evaluation.

Second, Lime's binding time annotations are fully integrated into the type system, which provides safe, modular checking. In contrast, previous systems for C were usually unsafe, or relied on aggressive interprocedural alias analysis to reason about side effects and reaching definitions. For example, in DyC, the programmer could annotate array contents as immutable, but the system could not verify these annotations.

In general, binding time analysis in traditional partial evaluation systems must perform aggressive, challenging interprocedural static analysis to compute a global division from annotations *e.g.*, [23,22]. Experience has shown that such analysis often fails to be sound for real-world Java programs. Instead, the Lime language allows modular, sound type checking.

Additionally, Lime restricts the static division to oblivious code sequences [11], which can include holes for values, but where dynamic values cannot affect control flow. We are not aware of any previous type systems or languages which specifically target oblivious algorithms.

10 Conclusion

We have presented language constructs that allow the programmer to use the full power of an object-oriented language to construct stream graphs, yet still allow the compiler to extract static shape information needed to compile to an FPGA. This work was motivated by constraints when compiling to hardware, which makes knowledge of static structure a necessity, and not just an optimization. However, our language approach might prove useful in other domains which would benefit from analysis of complex static structures built with general purpose language abstractions.

References

1. Thies, W.: Language and compiler support for stream programs. PhD thesis, Massachusetts Institute of Technology (2009)
2. Hirzel, M., Gedik, B.: Streams that compose using macros that oblige. In: Proceedings of the SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 141–150 (2012)
3. Chambers, C., et al.: FlumeJava: Easy, efficient data-parallel pipelines. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 363–375 (2010)
4. Plavec, F., Vranesic, Z.G., Brown, S.D.: Towards compilation of streaming programs into FPGA hardware. In: Forum on Specification and Design Languages, pp. 67–72 (2008)
5. Gokhale, M., Stone, J., Arnold, J., Kalinowski, M.: Stream-oriented FPGA computing in the Streams-C high level language. In: Field-Programmable Custom Computing Machines, pp. 49–56 (2000)
6. Mencer, O.: ASC: A stream compiler for computing with FPGAs. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. 25, 1603–1617 (2006)
7. Hormati, A., Kudlur, M., Mahlke, S., Bacon, D.F., Rabbah, R.: Optimus: Efficient realization of streaming applications on FPGAs. In: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 41–50 (2008)
8. Auerbach, J., Bacon, D.F., Cheng, P., Rabbah, R.: Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In: OOPSLA, pp. 89–108 (October 2010)
9. Arnold, K., Gosling, J.: The Java Programming Language, 2nd edn. Addison-Wesley, New York (1998)
10. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1988)
11. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall International (1993)
12. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21, 5–19 (2003)
13. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XII, pp. 151–162. ACM, New York (2006)
14. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21, 666–677 (1978)
15. Buck, I., et al.: Brook for GPUs: Stream computing on graphics hardware. ACM Trans. Graph. 23, 777–786 (2004)
16. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Proceedings of the 11th International Conference on Compiler Construction, London, UK, pp. 179–196 (2002)
17. Mainland, G., Morrisett, G., Welsh, M.: Flask: staged functional programming for sensor networks. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 335–346 (2008)
18. Taha, W.M.: Multistage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute (1999)

19. Leone, M., Lee, P.: Dynamic specialization in the Fabius system. ACM Comput. Surv. 30 (1998)
20. Poletto, M., Engler, D.R., Kaashoek, M.F.: tcc: a system for fast, flexible, and high-level dynamic code generation. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 109–121 (1997)
21. Grant, B., et al.: DyC: an expressive annotation-directed dynamic compiler for C. Theor. Comput. Sci. 248, 147–199 (2000)
22. Consel, C., Noël, F.: A general approach for run-time specialization and its application to C. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 145–156 (1996)
23. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. TOPLAS 25, 452–499 (2003)

Higher-Order Reactive Programming with Incremental Lists

Ingo Maier and Martin Odersky

EPFL
{firstname.lastname}@epfl.ch

Abstract. Reactive programming with first class time-varying values as in Functional Reactive Programming (FRP) is a powerful paradigm for designing and implementing event-based applications. Existing implementations deal with simple values. Time-varying collections can only propagate whether they have changed or not but not what has changed. This is inefficient compared to fine-grained callback driven logic that propagates incremental changes. In this work, we present a framework, Scala.React, with reactive abstractions for event streams, time-varying values as well as an incremental reactive list. Our reactive lists support both first-order reactivity by means of composition similar to functional collections via `map`, `filter`, `fold` and alike, as well as higher-order reactivity with support for time-varying collection elements. The framework automatically propagates incremental changes and guarantees strong data consistency. We show in examples that our system is convenient to use and performs well.

1 Introduction

In contrast to batch mode systems, reactive systems constantly interact with their environment. They require substantial effort to process input, track data dependencies and produce output. Common examples of reactive systems are user interface applications, trading applications, simulations, games, embedded systems, and sensor networks. Traditionally, event handling is implemented in imperative languages in terms of callbacks. Reasoning about complex event logic and dynamic data flow in callback-driven code quickly becomes impossible. Various approaches to raise the level of abstraction and to simplify the implementation of reactive systems have been investigated for almost 50 years now [26].

A fairly recent approach, which greatly simplifies local reasoning about data flow, is functional reactive programming (FRP). It moves the burden of tracking data dependencies from the application programmer to a general reactive framework and takes care of propagating changes in the correct order. Common abstractions of FRP systems integrated into imperative programming languages are time-varying values (in the following called *signals*) and event streams [7,21,19]. An event stream represents an entity that emits events and can be composed functionally. We can, for example, create an event stream that emits all events from two input event streams. A signal, on the other hand, holds a value and

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 707–731, 2013.

can be composed using expressions from the host language which automatically change when one of the referred signals change. These abstractions are first-class, i.e., they can be used to abstract over them and a signal or event stream can hold or emit other signals and event streams.

Existing FRP implementations propagate absolute values, with no support for incremental changes in data structures. Many reactive applications, however, maintain time-varying lists of elements in memory. Simulations and games maintain lists of agents, a task manager a list of tasks, an address book a list of address entries, file managers directory contents, etc. These lists are constantly changed, queried and displayed as a whole or partially. In existing FRP implementations, such time-varying lists must be represented as signals of non-reactive collections. When combined with functional collection composition, this results in reevaluating entire collections even for the smallest changes. For an example, consider an address book application that manages a time-varying list of contact records with a name, address and phone number. If we map that list to a list of the corresponding names, e.g., for display in a list view, we don't want to recompute the entire names list when a single contact gets added to the contact collection. We rather want to just add the name of the new contact to the names list. We can take this idea one step further. In practice, names, addresses and phone numbers can change, i.e., they are time-varying. A contact entry can therefore be represented as a record of signals. In this case, we also want to perform the least amount of recomputation when an existing contact name changes – ideally just change the affected name in the names list.

1.1 Contributions

In this work, we develop a reactive programming framework, Scala.React, with support for functional reactive collection composition and propagation of incremental changes. We develop a reactive list data structure that supports higher-order reactivity. Our main contributions are in particular:

- We show how an incremental reactive list data structure that supports common combinators such `filter`, `map`, `fold` integrated into a functional reactive language simplifies the implementation of complex event logic and higher-order data flow involving time varying collections.
- We discuss the conceptual as well as practical challenges that arise in the definition and implementation of collection operations such as `filter`, `flatMap`, folds and in the support for higher-order reactivity. We introduce a binary tree of sequence segments, which automatically balances itself in reaction to changes and uniformly solves the problems we identified.
- We give time as well as space complexities for reactive lists and their operations and present a performance analysis that compares our reactive list with time-varying non-incremental lists.

2 Overview

In this section, we examine how we address the problems identified above in our reactive programming model with the help of a representative example.

Consider the implementation of an in-memory database of albums, artists and songs in a media player. An entry is of type `Album` which stores a list of tracks. The abstract base types are defined as follows:

```
abstract class Album {
  def title: Signal[String]
  def artist: Signal[String]
  def tracks: RSeq[Track]
  def category: Signal[String]
}
```

```
abstract class Track {
  def album: Album
  def number: Signal[Int]
  def title: Signal[String]
  def file: Path
}
```

Most album and track properties in a media player can be modified by the user. Therefore, they are of type `Signal[T]`, the base class for time-varying values of type `T` in Scala.React. We can create a label that shows the currently playing track using signal composition as follows:

```
val currentTrack: Signal[Track] = ... // the currently playing track
val playingLabel: Signal[String] = Signal {
  val t = currentTrack()
  "Playing " + t.title() + " by " + t.album.artist()
}
```

The resulting string signal `playingLabel` will change whenever any of its dependencies change. A signal created via syntax `Signal { expr }` establishes dependencies with all signals that were accessed in `expr` with function call syntax and reevaluates `expr` whenever they change. We say `expr` *applies* those signals accessed via function call syntax. In the example, `playingLabel` applies and therefore depends on the `currentTrack`, the current tracks `title` and the `artist` of its album. We could now use `playingLabel` to create a label view in the user interface, for example.

The album class contains a reactive list of `tracks`. It is of type `RSeq[Track]`, the base class for reactive sequences in Scala.React. We could have defined it as of type `Signal[Seq[Track]]`, using base type `Seq` for non-reactive collections from Scala's standard library. This would lead to the problems we identified in the introduction. For example, given a list of albums of type `Signal[Seq[Album]]`, to create a list of all album titles we would have to write:

```
val albums = new Var[Seq[Album]](Nil)
val albumTitles: Signal[Seq[String]] = Signal {
  albums() map { album => album.title() }
}
```

Class `Var[T]` implements a mutable signal of `T` and takes an initial value in its constructor – in this case an empty list. The resulting signal `albumTitles` would now depend on the `albums` signal and each album's `title` and would create a new mapped sequence for every change in the albums list or in any of the album titles.

Obviously, this implementation would be very expensive (see Section 6). Instead, we store the list of albums in a reactive buffer which we can map directly:

```
val albums = new RBuffer[Album]
val albumTitles: RSeq[String] = albums sigMap { album => album.title() }
```

Class `RSeq[T]` is the base class for reactive sequences with elements of type `T`. Class `RBuffer[T]` is a subtype of `RSeq[T]` which exposes a mutable interface just like `Var` does for signals. Throughout this paper, we use the terms *reactive list* and *reactive sequence* interchangeably.

Operator `sigMap` works like the common map operator on immutable collections with an important difference: it tracks all changes in the input collection `albums`. It also establishes dependencies to all applied signals in the closure argument – every album's title in the example. The resulting `RSeq albumTitles` is now connected to the original `album` list and will automatically change, whenever `albums` changes, e.g., when an album is inserted or removed. On every change, `albumTitles` will incorporate changes and only those changes from its input list `albums`. For instance, if a new album gets appended to `albums`, `albumTitles` will only append the title of that album to itself. The reactive list `albumTitles` will also automatically change whenever any title of an album currently contained in list `albums` changes. We will discuss details on the precise effects of such changes in Section 4.

```
trait Seq[+A] {
  def size: Int
  def apply(idx: Int): A
  def contains[B >: A](a: A): Boolean
  def count(p: A=>Boolean): A
  def exists(p: A=>Boolean): Boolean

  def map[B](f: A => B): Seq[B]
  def filter(p: A => Boolean): Seq[A]
  def flatMap[B](f: A => Seq[B]): Seq[B]
  def ++[B >: A](that: Seq[B]): Seq[B]

  def fold[B >: A](z: B)(op: (B, B) => B): B
  def foldLeft[B](z: B)(op: (B, A) => B): B
  def foldRight[B](z: B)(op: (A, B) => B): B
}
```

Fig. 1. The main interface of a Scala standard collection sequence

There are many more operators defined in class `RSeq`. Figure 2 shows the most commonly used methods from `RSeq`. The interface largely mimics the interface of non-reactive `Seq` shown in Figure 1 for reference[1]. All methods in `RSeq` now return signals or reactive sequences instead of plain values or non-reactive

[1] The hierarchy is slightly simplified and some methods actually defined in base classes.

sequences reflecting the fact that anything can change. Some methods come in two versions, one with a `sig` prefix. This prefix indicates that in this version of the method, the argument function is allowed to apply signals. We call such a version *higher-order reactive* since it is parameterized by an (unknown and potentially varying) number of signals. Methods without a `sig` version depend only on a fixed and known number of dependencies and are therefore called *first-order reactive*. Figure 4 shows a few examples using different operations on `RSeq`. Note that we are free to use any Scala expression inside those queries.

```
trait RSeq[+A] extends Reactive[RSeqDelta[A], Seq[A]] {
  def now: Seq[A]
  def size: Signal[Int]
  def apply(idx: Int): Signal[A]
  def contains[B >: A](a: B): Signal[Boolean]
  def count(p: A=>Boolean): Signal[Int]
  def sigCount(p: A=>Boolean): Signal[Int]
  def exists(p: A=>Boolean): Signal[Boolean]
  def sigExists(p: A=>Boolean): Signal[Boolean]

  def map[B](f: A => B): RSeq[B]
  def sigMap[B](f: A => B): RSeq[B]
  def filter(p: A => Boolean): RSeq[A]
  def sigFilter(p: A => Boolean): RSeq[A]
  def flatMap[B](f: A => RSeq[B]): RSeq[B]
  def sigFlatMap[B](f: A => RSeq[B]): RSeq[B]
  def ++[B >: A](that: RSeq[B]): RSeq[B]
  def foldUndo[B](init: B)(op: (B, A) => B)(undo: (B, A) => B): Signal[B]
  def aggregate[B](init: B)(op: (B, A) => B)(combine: (B, B) => B): Signal[B]
  def sigAggregate[B](init: B)(op: (B, A) => B)(combine: (B, B) => B): Signal[B]
}
```

Fig. 2. Most commonly used methods in the base interface of reactive sequences

For reference, Figure 3 shows the mutable interface of an `RBuffer` which is the same as the interface for non-reactive `Buffer`s in Scala (not shown).

2.1 Evaluation Model

Before we will discuss individual collection operations, we give a conceptual overview of Scala.React's evaluation model and how it propagates changes[2].

Scala.React maintains a number of acyclic dependency graphs each encapsulated in their own *domain*. Domains can communicate asynchronously with each other, but propagation inside a single domain is synchronous. Asynchronous communication across domains is out of the scope of this paper – we will limit our view to that of a single domain. A domain proceeds in discrete, non-overlapping

[2] We will simplify details that are not relevant in the context of this work.

```
trait RBuffer[A] extends RSeq[A] {
  def insert(idx: Int, elems: A*)
  def update(idx: Int, a: A)
  def +=(a: A)
  def append(as: A*)
  def remove(idx: Int, count: Int)
  def remove(idx: Int)
  def clear()
}
```

Fig. 3. The mutable interface of reactive sequences

turns. Between turns, a domain receives updates to *source* nodes, which are nodes that do not depend on other nodes, such as the albums `RBuffer` from above. As soon as there is a pending update, a domain makes sure that a turn is scheduled. A turn consists of two phases, admission and propagation.

When a domain enters a turn, it starts with the admission phase, where all pending updates to source nodes are collected and added to a list of updated sources. The domain then enters propagation, where admitted changes are transitively propagated through the dependency graph, for example from the `RBuffer albums` to the `RSeq[Track] tracks` and then to the `RSeq[String] trackNames` in the examples from Figure 4.

Scala.React generalizes signals, event streams and reactive collections into *reactives* of type `Reactive[P,V]`. A reactive holds a value of type `V`, which can be obtained via method `now: V`. In a turn in which a reactive changes, it holds a pulse of type `P`, which represents a change. For signals, a pulse represents the new value, therefore `Signal[A]` is a subclass of `Reactive[A,A]`. Event streams only exist for their pulses. Their values never change. Therefore, `Events[A]` is a subclass of `Reactive[A, Unit]`[3]. A reactive list `RSeq[A]` has a standard `Seq[A]` as its current value and a pulse of type `RSeqDelta[A]`. The pulse of an `RSeq` is a delta to its previous value. We specify a delta of an `RSeq[T]` as an element of set Δ, defined as follows:

$$
\begin{aligned}
\Delta_{\text{Ins}} &= \{\texttt{Ins}_\Delta(i, e) \mid i \in \mathbb{N}, e \in \texttt{T}\} \\
\Delta_{\text{Rem}} &= \{\texttt{Rem}_\Delta(i, e) \mid i \in \mathbb{N}, e \in \texttt{T}\} \\
\Delta_A &= \Delta_{\text{Ins}} \cup \Delta_{\text{Rem}} \cup \{\varepsilon_\Delta\} \\
\Delta_C &= \{\texttt{Conc}_\Delta(\delta_1, \delta_2) \mid \delta_1, \delta_2 \in \Delta\} \\
\Delta &= \Delta_C \cup \Delta_A
\end{aligned}
$$

[3] Type `Unit` in Scala is equivalent to `java.lang.Void` in Java. We won't go further into event streams, since they are not relevant for this work beyond demonstrating the more general concept of reactives.

```
// all albums in the blues category
val bluesAlbums: RSeq[Album] = albums sigFilter { album =>
  album.category().toLowerCase == "blues"
}

// all blues albums by artist Eric Clapton
val bluesAlbumsCountByClapton: RSeq[Album] = bluesAlbums sigCount { album =>
  val artist = album.artist().toLowerCase
  artist == "clapton" || artist == "eric␣clapton" || artist == "clapton,␣eric"
}

// a list of all tracks
val tracks: RSeq[Track] = albums flatMap { _.tracks }

// a list of all tracks names, e.g., for use in a list view
val trackNames: RSeq[String] =
  tracks sigMap { t => "#" + t.number() + "␣" + t.name() }

// the total disk space occupied by all track files
val totalFileSize: Signal[Int] =
  tracks.foldUndo(0) { (res, track) => res + track.file.size }
                     { (res, track) => res - track.file.size }

// (One of) the album(s) with the fewest tracks.
val albumWithFewestTracks: Signal[Album] =
  albums.sigAggregate[Option[Album]](None) { minAlbum _ } { (result, album) =>
    result match {
      case Some(res) => Some(minAlbum(res, album))
      case _ => Some(album)
    }
  }

// helper function for the fewest tracks query
def minAlbum(a1: Album, a2: Album) =
  if(a1.tracks.size() < a2.tracks.size()) a1 else a2
```

Fig. 4. Some queries on the list of albums

A delta can either be an atom, the smallest unit of change, or a concatenation of other deltas. Atoms $\texttt{Ins}_\Delta$ and $\texttt{Rem}_\Delta$ represent element insertions and removals of an element e at index i. ε_Δ represents the null pulse, i.e., no change in the reactive list. We include ε_Δ in the set of deltas in order to make all helper functions below total, which will prove useful later. Note that delta concatenation is not commutative. For example, $\texttt{Ins}_\Delta(0, x)$ followed by $\texttt{Ins}_\Delta(0, y)$ is a different change than $\texttt{Ins}_\Delta(0, y)$ followed by $\texttt{Ins}_\Delta(0, x)$ for $x \neq y$.

When a reactive is evaluated during propagation, it evaluates its pulse and value based on the pulses and values of its dependencies. A signal simply evaluates expression `e` in signal constructor `Signal{ e }`. A reactive list evaluates its pulse based on which combinator it was created by. In the following, we will discuss how these pulses are evaluated to understand the complexity of different operations on reactive lists and eventually their implementation. Note that we abstract from dynamic dependency management and the propagation strategy in our descriptions. Because we include ε_Δ in the set of deltas, we can evaluate the pulse of a derived `RSeq` in any turn. Since the dependency graphs we create are acyclic, we can assume that a reactive is always evaluated after its dependencies. A pulse and value of a reactive r can be evaluated from current pulses and values of other reactives and the previous value of r.

3 The Concat Tree

In well-designed applications, the choice of data structures influences the design of the application and vice versa. The choice of data structure is influenced by factors such as expected access patterns and space consumption. In a higher-order reactive system like ours, access patterns are generally impossible to predict. Moreover, the choice of data structure in one reactive list has an effect on all dependent lists. Consider a list created by `sigMap` or `sigFilter`. An update of an element at a random position in the input list results in a random access into the dependent list and possibly in transitive dependents.

In the following, we will introduce a data structure that allows us to implement almost all operations on reactive lists in $O(\log n)$ or less time. We call the data structure a *concat tree*. A concat tree is a binary tree that recursively subdivides a list as shown in Figure 5 and automatically balances itself when the underlying list changes. The leaves split a list into non-overlapping segments without any gaps between segments. Each inner node represents the concatenation of its two children. The root therefore represents the entire list. Each node stores two child pointers, a parent pointer and a length, which is the number of elements in the list represented by the node. The length of an inner node is therefore the sum of the lengths of its children. It is important that nodes do not store their positions in the entire list.

There exist different variations of this tree in our system, all sharing the same base functionality. An `RBuffer` uses a mutable concat tree (which we call a *segment buffer*) that stores a dynamic array in each leaf. A segment buffer is a generalization of a dynamic array or more generally a gap buffer, with efficient

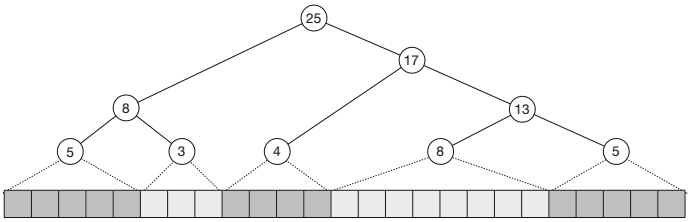


Fig. 5. A concat tree, recursively decomposing a sequence with leaves projecting segments from an underlying sequence. Nodes show lengths of their segments.

insertion and removal at random indices. Inserting or removing an element at a random list index i in a segment buffer of n elements and m leaves works as follows. We first search the leaf whose range contains i, starting from the root recursively choosing a child by comparing lengths. As we will see, a concat tree is balanced, therefore finding the corresponding leaf takes $O(\log m)$ time. We then insert or remove the element in the leaf segment, which takes $O(g)$ time, where g depends on the representation of the list segment. As this changes the leaf length, we follow parent pointers to the root, updating length values on the way, which takes $O(\log m)$ time. Note that if we would store positions in nodes, an insertion or removal in a leaf segment would invalidate positions of all following segments, making insertions and removals $O(m)$ time operations on average. Without the need to update node positions, insertions and removals take $O(g + \log m)$ time. For example, if leaf segments are dynamic arrays with length bounded by constant l, then $O(g) = O(l) = O(1)$ and total insertion/removal time is $O(\log m)$.

The other major use case of the concat tree in our system is to obtain a stable segmentation of an existing time-varying list so that a change in one segment does not invalidate other segments. Imagine the tree in Figure 5 subdivides an underlying list xs of unknown representation. Leaves do not store contents but by their length value and position in the tree represent or *project* a segment of xs. When elements gets inserted or removed from xs, the tree needs to adapt the length values and only those that have changed. For example, if xs inserts an element at index 9 in the list in Figure 5, we first find that this change affects the third leaf from the left in $O(\log m)$ time, just like we do for insertions into segment buffers. The leaf then needs to update its length value to 5, as well as its ancestors up to the root to 18 and 26. Because nodes do not store indices, we don't have to update other leaves and their ancestors. Therefore, adapting the concat tree to a single insertion or removal takes $O(\log m)$ time. We essentially just retraced the execution of an insertion into a segment buffer.

We often want a concat tree to split a list into segments not smaller than a threshold s_{min} and not larger than a threshold s_{max}. For a segment buffer, for example, if s_{max} is not bounded, insertion and removal become $O(n)$ operations. If s_{min} is too small, the tree becomes large and operations on the leaves will

benefit less from cache locality. Therefore, if a leaf length becomes larger than a given constant threshold s, the tree will split the leaf into two. Conversely, if a leaf length becomes less than s_{min}, the leaf is merged with a neighboring leaf. Of course, s_{min} and s_{max} and the exact split and merge strategy have to be chosen in such way that a split does not create leaves smaller than s_{min} or a merge does not create leaves larger than s_{max}. Continuous updates of a list can result in an imbalanced tree if not counteracted. Every split or merge can therefore result in a number of tree rotations to balance the tree. Scala.React's internal interfaces do not depend on a specific algorithm. Therefore, the precise balancing approach is implementation specific and can easily be changed. We require that rebalancing is an $O(\log n)$ operation, however, so that our $O(\log n)$ time guarantee for all operations can be upheld.

4 Incremental Sequences

In this section, we will describe individual operations, their complexities and how they are evaluated. For space reasons, we will not describe every operation in detail but concentrate on those operations that demonstrate the most important ideas.

Before we proceed, we define a few helper functions on list deltas. The size of a delta $size_\Delta : \Delta \to \mathbb{N}$ is defined as the number of atoms in it :

$$\begin{aligned} \text{size}_\Delta(\texttt{Conc}_\Delta(\delta_1, \delta_2)) &= \text{size}_\Delta(\delta_1) + \text{size}_\Delta(\delta_2) \\ \text{size}_\Delta(\delta) &= 1, \quad \delta \in \Delta_A \end{aligned}$$

Pulse composition $\diamond : \Delta \times \Delta \to \Delta$ combines pulses sequentially. It is defined so that null pulses do not occur in $\texttt{Conc}_\Delta$:

$$\delta_1 \diamond \delta_2 = \begin{cases} \delta_1, & \text{if } \delta_2 = \varepsilon_\Delta \\ \delta_2, & \text{if } \delta_1 = \varepsilon_\Delta \\ \texttt{Conc}_\Delta(\delta_1, \delta_2), & \text{otherwise} \end{cases}$$

Pulse composition is therefore an $O(1)$ operation. We define two transformation functions on deltas. Function $\text{map}_\text{i} : \Delta \times (\mathbb{N} \to \mathbb{N}) \to \Delta$ maps indices and keeps elements:

$$\begin{aligned} \text{map}_\text{i}(\texttt{Conc}_\Delta(\delta_1, \delta_2), f) &= \texttt{Conc}_\Delta(\text{map}_\text{i}(\delta_1, f), \text{map}_\text{i}(\delta_2, f)) \\ \text{map}_\text{i}(\texttt{Ins}_\Delta(i, e), f) &= \texttt{Ins}_\Delta(f(i), e) \\ \text{map}_\text{i}(\texttt{Rem}_\Delta(i, e), f) &= \texttt{Rem}_\Delta(f(i), e) \\ \text{map}_\text{i}(\varepsilon_\Delta, f) &= \varepsilon_\Delta, \end{aligned}$$

Function $\mathrm{map_e} : \Delta \times (\mathtt{T} \to \mathtt{T}) \to \Delta$ is similarly defined and maps elements but keep indices:

$$\begin{aligned}
\mathrm{map_e}(\mathtt{Conc}_\Delta(\delta_1, \delta_2), f) &= \mathtt{Conc}_\Delta(\mathrm{map_e}(\delta_1, f), \mathrm{map_e}(\delta_2, f)) \\
\mathrm{map_e}(\mathtt{Ins}_\Delta(i, e), f) &= \mathtt{Ins}_\Delta(i, f(e)) \\
\mathrm{map_e}(\mathtt{Rem}_\Delta(i, e), f) &= \mathtt{Rem}_\Delta(i, f(e)) \\
\mathrm{map_e}(\varepsilon_\Delta, f) &= \varepsilon_\Delta,
\end{aligned}$$

Function $\mathrm{translate}_\Delta : \Delta \times \mathbb{N} \to \Delta$ moves indices by a constant:

$$\mathrm{translate}_\Delta(\delta, j) = \mathrm{map_i}(\delta, i \mapsto i + j)$$

It is easy to see that mapping indices takes $O(k \cdot g)$ time, where k is the size of the input delta and $O(g)$ the complexity of the given map function f. For translation, f takes $O(1)$ time, therefore translation takes $O(k)$ total time.

4.1 Mutating `RBuffer`s

The mutable interface of an `RBuffer` may be accessed only during the admission phase of a turn. An `RBuffer` stores a buffer for its value internally. Individual operations compose the pulse of their `RBuffer` in a straightforward way. Method

```
def remove(idx: Int)
```

for example, retrieves the current value v at index `idx`, e.g., via `v = now(idx)`, set the current pulse δ to $\delta = \delta \diamond \mathtt{Rem}_\Delta(\mathtt{idx}, \mathtt{v})$, and updates its internal buffer. Setting the pulse during admission takes $O(k)$ total time, where k is the size of the delta. Updating the value depends on the underlying buffer implementation. For a segment buffer, this takes $O(k \log n)$ time, which therefore is also the total time of admitting a delta for an `RBuffer`.

4.2 Mapping

Method `map` maps elements from one list to elements in an output list and is defined in `RSeq[A]` as follows:

```
def map[B](f: A => B): RSeq[B]
```

The resulting list's pulse δ is solely based on the pulse δ' of the original list: $\delta = \mathrm{map_e}(\delta', \mathtt{f})$. Processing deltas for `map` therefore takes $O(\mathrm{size}_\Delta(\delta') \cdot g)$ time, given `f` takes $O(g)$ time. Internally, a `mapped` list maintains a segment buffer as its value, which is updated for each pulse in $O(\log n)$ time. Updating pulse and value hence takes $O(\mathrm{size}_\Delta(\delta') \cdot g \cdot \log n)$ time.

4.3 Concatenation

Method ++ concatenates two lists and is defined in `RSeq[A]` as follows:

```
def ++[B >: A](that: RSeq[B]): RSeq[B]
```

The pulse δ of the resulting list is a combination of the current pulses δ_1, δ_2 of the left and right input lists and the current size s of the left input list: $\delta = \delta_1 \diamond \text{translate}_\Delta(\delta_2, s)$. Processing pulses for ++ therefore takes $O(\text{size}_\Delta(\delta_2))$ time. The value is a simple wrapper that delegates to the values of the two inputs and does not need to be updated.

4.4 Reversible Folds

Most common operations on non-reactive collections can be implemented in terms of folds (also known as *catamorphisms* [20]). Folds fix the structure of the recursion process and leave the initial value and the applied operation open. The recursion process is sequential; left and right fold iterate over a collection from left to right or vice versa, applying a function to each value and the accumulated result starting from an initial value. In the context of incremental collections, their sequential nature makes them seem out of place. Consider changing a value in the middle of the original list. In order to update the result of a left fold, we need to traverse the entire list, making left folds an $\Omega(n)$ time operation even for the smallest change. We can however put different constraints on the supplied operations and obtain a more efficient way to incremental folding. The most efficient version is `foldUndo`, which is defined in `RSeq[A]` as follows[4]:

```
def foldUndo(init: A)(op: (A, A) => A)(undo: (A, A) => A): Signal[A]
```

It takes an initial value `init` of type `A`, an operation `op` that is associative and commutative, and an *undo* operation for which the following must hold:

$$b \ \texttt{op} \ a \ \texttt{undo} \ a = b, \quad \forall a, b \in \texttt{A}$$

Because of associativity and commutativity of `op`, we can reorder applications of `op` at will. Because of the above constraint on `undo`, we can "remove" an element from the result as follows:

$$\begin{aligned} a \ \texttt{op} \ \ldots \ \texttt{op} \ x \ \texttt{op} \ \ldots \ \texttt{op} \ z \ \texttt{undo} \ x &= a \ \texttt{op} \ \ldots \ \texttt{op} \ z \ \texttt{op} \ x \ \texttt{undo} \ x \\ &= a \ \texttt{op} \ \ldots \ \texttt{op} \ z \end{aligned}$$

In practice, $(A, \texttt{op})$ often forms a commutative monoid with identity element `init`.

The signal resulting from `foldUndo` performs a left fold with `op` on creation to obtain its initial value. It then evaluates its current pulse (and therefore value)

[4] In practice, we can put weaker constraints on the types to obtain the signature `def foldUndo[A](init: B)(op: (B, A) => A)(undo: (B, A) => A): Signal[B]`. This complicates the algebraic properties of `op` and `undo`, however. We will restrict the discussion to the simple case.

from its previous value $v \in \texttt{B}$ and the current pulse $\delta \in \Delta$ of the sequence according to function $f : \texttt{B} \times \Delta \to \texttt{B}$ defined as follows:

$$\begin{aligned} f(v, \texttt{Conc}_\Delta(\delta_1, \delta_2)) &= f(f(v, \delta_1), \delta_2) \\ f(v, \texttt{Ins}_\Delta(i, e)) &= v \ \texttt{op} \ e \\ f(v, \texttt{Rem}_\Delta(i, e)) &= v \ \texttt{undo} \ e \\ f(v, \varepsilon_\Delta) &= v \end{aligned}$$

It easy to see that f processes deltas δ in $O(\text{size}_\Delta(\delta))$ time. We can now write, for example:

```
val sum: Signal[Int] = xs.foldUndo(0){_ + _}{_ - _}
```

This sums all elements to obtain an initial value. For every element that is inserted into `xs`, its value is added to the resulting signal's value. For every element that is removed, its value is subtracted from the resulting signal.

Examples. Method `size` is implemented in terms of `foldUndo`:

```
def size: Signal[Int] = foldUndo(now.size) { (x,y) => x+1 } { (x,y) => x-1 }
```

The initial value is the size of the list in the turn in which `size` is called, adding and subtracting one when elements get inserted or removed. Operator `exists` or `contains` are implemented in terms of a `count` operation, using `foldUndo` as follows:

```
def count(p: A=>Boolean): Sig[Int] =
  foldUndo(0) { (res, x) => if(p(x)) res+1 else res }
              { (res, x) => if(p(x)) res-1 else res }

def exists(p: A=>Boolean): Sig[Boolean] = {
  val c = count(p)
  Sig { c() > 0 }
}
```

We are using `foldUndo` to keep track of the element count that fulfills the predicate and then use a signal expression to convert it to a boolean signal.

4.5 Associative Folds

Not all operations commonly used in folds are both associative and commutative or are reversible. Binary minimum or maximum, e.g., is associative and commutative but is not reversible. String concatenation is associative but not commutative. Operations that are at least associative are in fact very common. Parallel collection frameworks often rely on precisely this fact to efficiently parallelize the folding process [24,25]. The signature of our associative fold operation is as follows:

```
def aggregate[B](init: B)(op: (B, A) => B)(combine: (B, B) => B): Signal[B]
```

The structure of such a fold recursively splits a list into segments as shown in Figure 5. This tree can be modelled as an abstract data type of the form:

$$\texttt{Tree} = \texttt{Seg}(x_0, ..., x_n) \mid \texttt{Conc}(xs, ys)$$

The fold operation f used by `aggregate` uses a left fold for the leaves and has a binary recursive structure for the inner nodes:

$$\begin{aligned} f(\texttt{Seg}()) &= \texttt{init} \\ f(\texttt{Seg}(x_0, ..., x_n)) &= ((x_0\ \texttt{op}\ x_1)\ \texttt{op}\ ...)\ \texttt{op}\ x_n \\ f(\texttt{Conc}(xs, xs)) &= f(xs)\ \texttt{combine}\ f(ys) \end{aligned}$$

The leaves could be modelled as single elements, which would simplify the structure of the fold. Unrolled segments as above, however, have multiple advantages in practice. We can customize the depth of the tree, use CPU caches more effectively, and as we will see below, customize the granularity of higher-order reactive operations.

In order to be able to react to changes in the original reactive list, the list resulting from `aggregate` reifies the above data flow graph of an initial run of the fold operation – in a conceptually similar way to how the core reactive framework stores a dependency graph, but more specialized to the tree structure. The structure of that tree is exactly the structure of our concat tree from Section 3. In the `aggregate` version of the concat tree, every node stores not only a length but also a value that is an intermediate result of the fold operation[5]. The root's value is therefore the final result. Leaf segments do not store elements directly, but merely project a slice from the underlying list to fold. The slice is determined by the segment length and its position in the tree.

We now go through an example that describes how the aggregated signal reacts to changes in the input list. Consider the input list inserts an element as shown in Figure 5. In order to update the fold result, we first find that insertion affects the third segment (of length l=4), update lengths and potential rebalance in $O(\log m)$ time, where m is the number of leaves, as described in Section 3. Given an `op` that runs in $O(g)$ time, we then left fold the segment in $O(g \cdot l)$ time. Given a `combine` that runs in $O(h)$ time, the result is then propagated up the tree in $O(h \cdot \log m)$ time, reusing intermediate fold results from previous runs. The total operation therefore has an average time complexity of $O(g \cdot \bar{l} + h \cdot \log m)$, where $\bar{l}$ is the average segment length. This process is shown in Figure 6. For composite deltas, we can naively repeat this process for every atom, leading to $O(k \cdot (g \cdot \bar{l} + h \log m))$ average time complexity, where k is the size of the delta. Since segment lengths are always constant bounded in our implementation, updating the result of an `aggregate` signal takes $O(k \cdot \log n)$ time for $O(1)$ `op` and `combine` operations.

[5] This is sometimes called a *monoid-cached* tree [15,25], since the `combine` operation together with element `init` usually form a monoid.

Even though not conceptually, but in practice, we can process composite deltas faster by evaluating all leaves first and then propagate up the tree in topological order. This is in fact how our list is implemented as we show in Section 5.

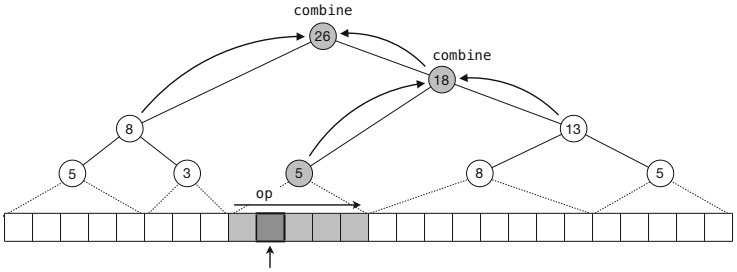


Fig. 6. Propagating a value up to the root. Reevaluated nodes are shaded.

Reversible folds and associative folds or more generally incrementally propagating changes up the tee as described above are the keys to efficient incremental reactive programming with lists. Obtaining incremental operations on reactive lists from batch operations on non-reactive collections essentially means translating linear recursion of sequential folds to tree recursion of associative folds or ideally eliminating recursion with reversible folds – replacing $O(n)$ with $O(k \cdot \log n)$ or $O(1)$ operations. The tree data structure above comes with a $O(\log m)$ cost in memory, though.

4.6 Higher-Order Reactivity with `flatMap`

Until now, all collection operations we discussed were first-order reactive in that there was a fixed number of reactive inputs. As we have seen in the overview, our system also supports higher-order reactivity in that reactive collections (in fact any reactive) can contain other reactives. One example is `flatMap`, which takes a function from the list element type to a reactive list:

```
def flatMap[B](f: A=>RSeq[B]): RSeq[B]
```

The resulting sequence therefore depends not only on the input collection, but also on a varying number of lists obtained from argument function `f`.

Interestingly, `flatMap` and in fact many higher-order operations, can be implemented in a similar way to an associative fold using a concat tree. This time, the leaves of the tree store the lists obtained from `f`, as depicted in Figure 7 and not just length values of an input list. For every change in the input list, the implementation applies argument function `f` in $O(g)$ time, inserts or removes a leaf in the tree, potentially followed by a rebalancing operation, which both take $O(\log n)$ time, where n is the size of the input list. Therefore, a single insertion or removal in the input list results in an $O(g + \log n)$ time update of the current list value of the `flatMap`ped sequence.

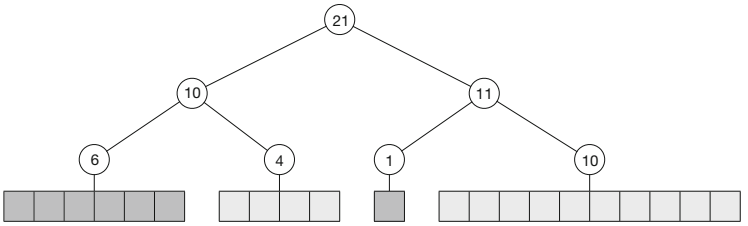


Fig. 7. A concat tree for `flatMap`. Leaves store collections.

The nodes of the concat tree store deltas as values. For changes in the leaf lists, we perform a tree traversal as for `aggregate` in Figure 6. Function `op`, applied sequentially at the leaves, simply takes the current delta of the corresponding `RSeq`, whereas function `combine`, combining results at the inner nodes, takes the delta δ_1 and length l from the left child and the delta δ_2 from the right child and composes them as the concat (`++`) operation, i.e.:

$$\delta_1 \texttt{ combine } \delta_2 = \delta_1 \diamond \text{translate}_{\Delta}(\delta_2, l)$$

In this fashion, the delta indices are stepwise translated and on reaching the root eventually reflect the absolute index into the resulting `flatMap`ped list. We have already seen that this `combine` operation takes $O(1)$ time, applied $O(\log n)$ times to all nodes on the path from the leaf to the root. Therefore, updating the pulse in this fashion takes $O(\log n)$ time. In total, propagating a delta from the input to the `flatMap`ped sequence takes $O(k \cdot (g + \log n))$ time, where k is the size of the delta.

4.7 Higher-Order Reactivity with Time-Varying Elements

Since we want to be able to deal with collections that contain signals (directly or indirectly), collection combinators can semantically take function arguments that return signals:

```
def map[B](f: A=>Signal[B]): RSeq[B]
def filter(f: A=>Signal[Boolean]): Signal[Boolean]
def aggregate[B](init: B)(op: (B, A) => Signal[B])
                         (combine: (B, B) => Signal[B]): Signal[B]
```

A sequence mapped with the `map` function defined above would not only depend on the input list but also on the signals defined by argument function `f`. We have seen examples of this in Section 2, where we have also seen that we use signal applications in argument functions. This is because the above definitions have a big disadvantage. Consider the following, hypothetical filter operation:

```
class Person(name: Signal[String], address: Signal[String])
val persons: RSeq[Person] = ...
persons filter { p => Signal { p.name().startsWith("a") } }
```

This creates a new signal, i.e., new node in the dependency graph, for every person that gets added and removed to the collection. This is very fine-grained and can lead to large dependency graphs. What is worse is that some of these signals are used just once but are not disposed before the next garbage collection cycle[6]. This is a general problem with higher-order reactivity in FRP and a variation of the dangling dependent problem of the functional reactive drag operation as discussed in [19].

The above operations therefore come in pairs as we have seen in Figure 2, one first-order reactive version and one higher-order reactive version for which the argument functions are explicitly allowed to call `Signal.apply`. The implementation of those methods use the same mechanism as signals to establish dependencies, but have control over the granularity. They create one dependent per leaf segment of the original collection, which will be notified whenever the segment is modified or signals change that the argument functions have accessed during the evaluation of that segment. This way, the number of dependents created by a higher-order reactive list operation is reduced and stays constant if the list size stays roughly constant, even if elements get inserted and removed constantly. Details of this are implementation specific and discuss in Section 5.

Examples. Similar to how `exists` is implemented in terms of `count`, which in turn is implemented in terms of `foldUndo`, `sigExists` is implemented in terms of `sigCount`, which in turn is implemented in terms of `sigAggregate` as follows:

```
def sigCount(p: A=>Boolean): Signal[Int] =
  sigAggregate(0) { (res, x) => if(p(x)) res + 1 else res } { _ + _ }

def sigExists(p: A=>Boolean): Signal[Boolean] = Signal {
  val c = sigCount(p)
  Signal { c() > 0 }
}
```

A change in a signal applied in the predicate will invalidate a segment in the underlying `sigAggregate` tree and eventually propagate up the concat tree.

5 Implementation

We already gave an overview of the general mode of operation in Section 2.1. We will now discuss the most important aspects of our implementation in reactive lists and relevant parts of the general reactive framework.

5.1 Establishing Signal Dependencies

Some reactives, such as the from the first-order reactive `map`, `foldUndo`, or `++` sequence combinators, know their dependencies since they are explicitly supplied

[6] Internally, dependencies are stored in weak references to ensure that a dependency does not prevent a reactive from being collected.

as arguments. For signals created with syntax `Signal { ... }` and higher-order reactive combinators such as `sigMap` or `sigAggregate`, however, dependencies are supplied implicitly via signal application. We now give a quick overview how this works. The framework maintains a stack of dependents that is used by signals and reactives from other higher-order combinators. Before such a reactive evaluates a closure that is allowed to use signal application, it first pushes itself onto the stack as follows:

```
def evaluate[A](dependent: Node, closure: ()=>Unit): A = {
  dependentStack.push(dependent)
  try { closure() }
  finally { dependentStack.pop() }
}
```

where `dependent` is the node that is about to evaluate `closure` (class `Node` is an internal generalization of class `Reactive` and simply represents any node in the dependency graph). Signal application, which can be used in argument `closure` above, is defined in `Signal[V]` as follows:

```
def apply(): V = {
  checkTopology()
  subscribe(dependentStack.top)
  validate()
  getValue
}
```

It registers the top of the stack as a dependent, i.e., a node which will be notified when the enclosing signal changes. It then validates the signal and returns its current value. When a signal `sig` defined as `Signal{ a() + b() }`[7] is evaluated by calling `evaluate(sig, { a() + b() })`, method `apply` in signals `a` and `b` automatically establish dependencies between `a`, `b` and `sig`. This approach allows clients to use any language expression in signals and higher-order reactives. We need to use a *stack* of dependents, because method `validate` above can call `evaluate` recursively.

5.2 Data Consistency and Topology Mismatch

A domain maintains data consistency during a propagation turn, i.e., makes sure that no reactive can access the current state of another reactive that might have to be reevaluated because of a change during admission. This is an important invariant that enables local reasoning about reactive data-flow. Consider the following example:

```
def alwaysTrue_?(s: Signal[Boolean]) = {
  val notS = Signal { !s() }
  Signal { s() || notS() }
}
```

Given any input signal `s`, the function is rightfully expected to return a signal that is always `true`. However, if the resulting signal is updated before `notS` is

[7] Syntax `a()` is shorthand for `a.apply()` in Scala.

updated on a change in `s`, it might evaluate to `false`. This violation of data consistency is commonly referred to as a *glitch*.

In order to prevent glitches, nodes are evaluated in topological order by assigning levels to each node and traversing the graph from a priority queue. A source always has level 0, and the system maintains the invariant that a dependent has a strictly greater level than all of its dependencies. This ensures that no node is updated after any of its dependent nodes has been updated.

As we have seen, signals as well as collection operations can access arbitrary other signals during evaluation. Therefore, the topological order of the dependency graph can change during propagation. This can give rise to what we call a *topology mismatch*: a reactive r can access a node that has a level greater or equals its own level. Since this would violate data consistency, any such access is intercepted by the engine, stopping evaluation of r, increasing its level and those of its dependents if necessary and restarting r's evaluation once the propagation loop reaches the new level. It is also necessary that the levels of all nodes that transitively depend on r are adapted. In practice, topology mismatches are sufficiently rare and don't affect performance noticeably. The abortion mechanism is implemented in method `checkTopology` in the implementation of `apply` above by means of throwing an exception. Closure evaluation in `evaluate` is therefore wrapped in a try-finally block. The exception is finally caught by the propagation loop of the domain which performs the necessary level changes.

Since many reactives are largely opaque to the engine, it is the reactive's responsibility to ensure that this partially duplicate evaluation does not have adverse effects, such as applying pulses twice. For this reason, an `RSeq` needs to take special care to avoid redundant evaluation as discussed below.

5.3 The Concat Tree

We have seen the concat tree being used for three purposes: as the backing structure of a segment buffer, to partition a list for associative folds and to partition a list for higher-order combinators. The last use case needs further explanation.

Every leaf in a concat tree created by a higher-order reactive operation stores a dependency node in the domain's graph, which is used as the dependent in a call to `evaluate` when left folding its corresponding segment, establishing dependencies with any signal applied in the fold operation. When the dependency node of the leaf is notified by the domain of a change in one of its dependencies, it adds itself to an invalid queue of the tree, which in turn notifies the enclosing reactive of a change. When the enclosing reactive evaluates itself, it asks the tree to propagate from invalid leaves to the root of the tree.

When processing invalid leaves and folding a segment with an operation that can contain signal applications, the enclosing reactive has to prepare for the fact that a topology mismatch can happen. The concat tree therefore tracks its progress so that an `RSeq` can safely be aborted and resumed where it aborted when a mismatch occurs. For example, `sigAggregate` and others maintain a queue of invalid segments, which are updated one after the other so that after a level

mismatch the queue still contains the remanding invalid segments. How other types of reactives handle topology mismatches can be found in [19].

Balancing and Propagating. Our current implementation uses an AVL tree to balance a concat tree. We chose an AVL tree because it stores heights on its nodes. Heights define a topological ordering with leaves being the smallest elements, which we can use for propagation. Consider the tree in Figure 6. If in addition to the depicted change, there is an additional change in leaf with length 8, the propagation mechanism has to make sure that node with length 13 is updated before its value is obtained during evaluation of node with length 18. This is a glitch as discussed in Section 5.2. Propagating in topological order from lower to higher heights makes sure that glitches cannot occur.

When an AVL tree rebalances itself, it performs a number of tree rotations. The concat tree needs to make sure that length values are updated accordingly. Variants that propagate values from the leaves to the root, such as the trees used by `aggregate` or `flatMap`, also need to reevaluate parts of the subtree. This is shown in Figure 8. Since node R has different children after the rotation, its value needs to be reevaluated, which then causes P and all other ancestors to be evaluated. The lengths only change for the two nodes about which the rotation was performed, i.e., P and R in the example.

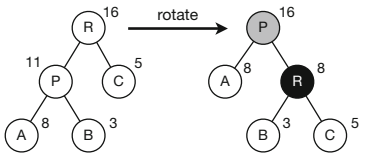


Fig. 8. A right rotation, showing lengths. Node R is immediately invalidated, node P is invalidated by R during propagation.

6 Discussion and Evaluation

A reactive sequence has two advantages over a signal of a non-reactive sequence. First, it is more convenient to use. Operations can be invoked on the sequence directly instead of wrapping calls in a signal constructor. Secondly, it can be much more efficient, both in terms of memory footprint as well as propagation times, but can also have lower baseline performance due to maintenance of a larger dependency graph. In the following, we will evaluate how performance is affected by multiple factors. All benchmarks are run with Scala 2.10 on Java Hotspot 7. We followed established performance measurement methodologies [11] and average results over multiple VM invocations with sufficient warmup iterations.

An `RSeq` propagates changes incrementally, which allows new operations such as reversible folds, reducing propagation times from $O(n)$ to $O(k)$, where k is the size of a delta. Other operations reduce running times from $O(n)$ to $O(k \log n)$ by keeping the dependency graph in memory and only reevaluating invalidated data-flow paths. Figure 9 shows the effect of using `foldUndo` compared to a signal

of a high-performance immutable vector from Scala's standard library. We also include signals folded with `aggregate` (segment size = 32) even though the operation satisfies `foldUndo`'s requirements and does not access signals. This gives us an indication of the overhead of tree propagation as well as dependency creation. We show two tests, one (1) updating a single element at a random position in the source `RBuffer`, and another (2) removing a single element at a random position and inserting a random element at another random position. A single observer on that signal tracks the value in the folded list or signal. As expected, we can see that `foldUndo` beats every other case by an increasing margin for lists larger than about $n = 15$. Left folding vectors with updated elements stays ahead of all other cases for a small number of elements, due to the smaller dependency graph and less indirections. Insertion and removal at random indices is less efficient for a vector than for an `RBuffer`, however, which results in lower performance than for `aggregate`, even for very short lists. Both `aggregate` cases beat case (1) for `foldLeft` with increasing margin from starting at around 100 elements. The `aggregate` cases slowly diverge because case (2) results in unbalanced trees. The continuous rebalancing results in only moderately slower performance.

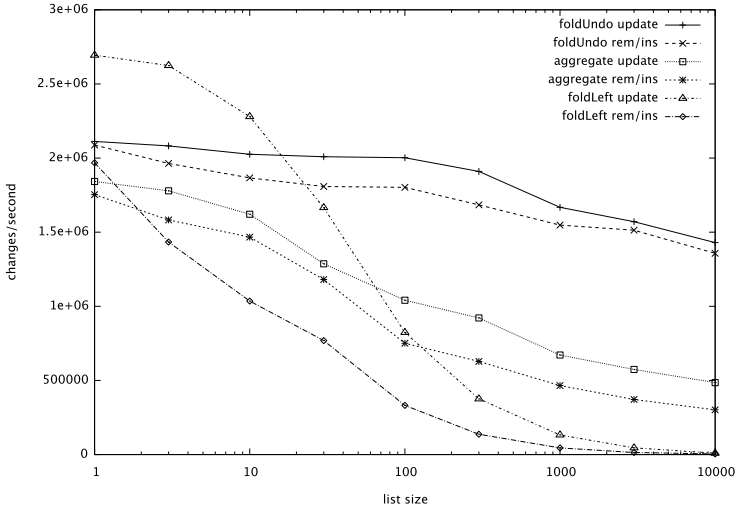
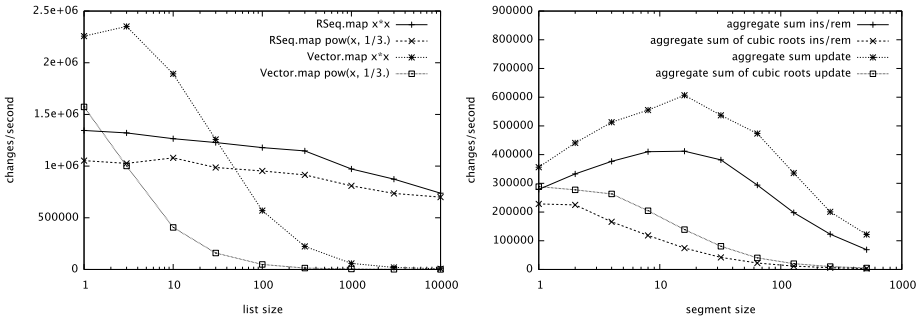


Fig. 9. Sum with different folds

Figure 10a shows how different `map` arguments affect performance. The graph shows numbers for mapping to squares compared to mapping to cubic roots. The test consists of updating a single element at a random position. For the inexpensive squares operation, the incremental nature of `RSeq.map` pays off for lists longer than around $n = 30$ elements, whereas for the relatively expensive cubic root operation, it already starts paying off for lists with more than 3 elements. Again, the margins keep increasing for greater n, reaching about factor 100, respectively 1500 for $n = 10000$.

Finally, the segment size of the concat tree affects performance in two ways. Smaller segment sizes lead to smaller linear folds at the leaves resulting in fewer reevaluations. On the other hand, smaller segment sizes also result in larger graphs, i.e., higher memory footprint and more work for change propagation as well as more tree balancing. Figure 10b shows the effects of varying segment sizes for aggregated lists of elements of size 3000. For inexpensive operations such as simple sums, very small segment sizes are not beneficial. Dependency and tree management as well as cache effects affect performance noticeably. More expensive operations such as summing up cubic roots, already benefit from very small segment sizes at the cost of larger trees and dependency graphs.



(a) Mapping lists with different f (b) `aggregate` with different segment sizes

Fig. 10. Performance of `map` and `aggregate` with varying parameters

7 Related Work

Our main framework is based on an extension of functional reactive programming, which has been explored in many variations [9,16,23,7,21]. Our implementation based on a topological sort is most closely related to FrTime [7] and Flapjax [21]. Details are very different, however. The way we automatically deal with any host language expression is closely related to the constraint system Amulet [28]. We are not aware of an FRP implementation with support for incremental updates or custom nested propagators as in our reactive list implementation in terms of the concat tree.

In terms of incrementality, the most closely related work is self-adjusting computation which is concerned about taking existing non-incremental programs and convert them into incremental versions with minimal manual work. Adaptive function programming (AFP) [2] is one example. AFP logs and replays arbitrary operations on mutable variables using a time stamped dependency graph. This allows to implement time-varying data structures in AFP. Dependency management, however, is very fine-grained, potentially creating very large dependency graphs, which makes it less suitable for implementing efficient incremental collections with time-varying elements. Recent work [1] has addressed the issue of

granularity, by tracing the effects of operations on arbitrary data types. Our work, on the other hand, focusses on specially designing a functional reactive data structure and its operations to be amenable for incremental computation, reusing an existing higher-order FRP propagation framework. Burckhardt et al. extend the parallel programming model of concurrent revisions[4] to incremental parallel programming in [5]. It is a form of self-adjusting computation that supports a notion of adaptive granularity using the fork/join style revision model.

Incoop [3] is an incremental MapReduce framework built on top of Hadoop which is the most closely related self-adjusting computation system in terms of data structure design. In contrast to our work, Incoop's focus is on distributed, parallel programming in the MapReduce model, whereas we focus on in-memory collections with more general functional combinators such as folds. Incoop, as well as Scala.React, rely on a stable partitioning scheme to allow for general incremental operations and granularity control. For this purpose, Incoop extends Hadoop's distributed file system to split the input into segments. Stability of those segments is achieved by a content-based splitting mechanism operating on binary data, whereas in Scala.React, stability is a achieved by fixing segment capacities and exposing stable segment references (note that a content-based partitioning for in-memory structures would likely be less efficient than our stable segment scheme). In both approaches, partitioning isn't perfectly stable, i.e., in some situations, the segments are rearranged: in Incoop for large changes in the input data, in Scala.React when a segment shrinks or grows beyond a threshold and is subsequently combined or split. Ultimately, self-adjusting computation systems focus on obtaining efficient algorithms in the presence of incremental updates, starting from the traditional batch programming model with explicit mutations. In contrast, our focus is on integrating incremental data structures into a higher-order reactive programming model to simplify complex event logic, establishing and maintaining functional dependencies instead of tracing effects.

Another related area is incremental view maintenance in databases [6,13,12,14,17]. Gluche et al. [12] formalize collection composition and decomposition into segments in terms of monoids, where the monoid operation is concatenation for lists and union for sets and bags. Incremental operations over collections are then expressed in terms of monoid homomorphisms. Our concat tree is essentially an implementation of such a decomposition. Fegaras and Maier [10] give a similar formalism extended with a calculus of monoid comprehensions which allows to combine collections of different types, for example, mapping lists into sets. They model indexed sequences as sets of value-index pairs, which is different from our representation. In contrast to these formalizations, we focus on a practical implementation of an in-memory data structure and discuss issues of data structure design, tree balancing and delta propagation.

Related to incremental view maintenance is research on query caching and incrementalization, for example, for XQuery [8], JQL [27] or general object languages [18]. The work in [8] on XQuery is different from ours in that it works on structured XML data. The work in [27] on JQL is different from ours in that it caches queries and their results for repeated similar queries on data that has

been updated since the last query. In contrast to our work, it does not work with folds. Liu et al. [18] show how to automatically incrementalize basic operations on sets but also does not work with folds. These systems are more related to adaptive programming in that they focus on obtaining efficient programs from existing non-incremental ones. Therefore, the same distinction to our work applies: in contrast to these systems, we expose reactivity to clients in form of our reactive sequence integrated into a broader reactive framework with abstractions for time-varying values (signals).

8 Conclusion

We have presented the first higher-order functional reactive data structure, integrated into a system with high-order reactive programming abstractions such as time-varying values and event streams, substantially improving the expressiveness and efficiency of the system.

Our current implementation can be further optimized, mostly by reducing the number of indirections. We hope to bring down constant factors to make the incremental sequence competitive with signals of non-incremental sequences for even smaller sizes and less expensive operations. We are also currently applying the techniques learned in this work to implement reactive sets and maps. As hash table based implementations rehash when the size of the table changes, they do not allow stable segmentation like a sequence does. Our set and map implementations are therefore based on hash tries [22].

Our implementation of Scala.React can be downloaded from the first author's homepage at `http://lamp.epfl.ch/~imaier`.

Acknowledgements. We would like to thank our anonymous reviewers for their valuable feedback.

References

1. Acar, U.A., Blelloch, G., Ley-Wild, R., Tangwongsan, K., Turkoglu, D.: Traceable data types for self-adjusting computation. In: PLDI (2010)
2. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM Trans. Program. Lang. Syst. (2006)
3. Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U.A., Pasquin, R.: Incoop: Mapreduce for incremental computations. In: SOCC (2011)
4. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: OOPSLA (2010)
5. Burckhardt, S., Leijen, D., Sadowski, C., Yi, J., Ball, T.: Two for the price of one: A model for parallel and incremental computation. In: OOPSLA (2011)
6. Ceri, S., Widom, J.: Deriving production rules for incremental view maintenance. In: VLDB (1991)
7. Cooper, G.H., Adsul, B.: Embedding dynamic dataflow in a call-by-value language. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 294–308. Springer, Heidelberg (2006)

8. El-sayed, M.F., Mani, M., Shanmugasundaram, J.: Incremental maintenance of materialized XQuery views. In: ICDE (2006)
9. Elliott, C., Hudak, P.: Functional reactive animation. In: ICFP (1997)
10. Fegaras, L., Maier, D.: Optimizing object queries using an effective calculus. ACM Trans. Database Syst. (2000)
11. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. In: OOPSLA (2007)
12. Gluche, D., Grust, T., Mainberger, C., Scholl, M.H.: Incremental updates for materialized OQL views. In: Bry, F. (ed.) DOOD 1997. LNCS, vol. 1341, pp. 52–66. Springer, Heidelberg (1997)
13. Griffin, T., Libkin, L.: Incremental maintenance of views with duplicates. ACM SIGMOD (1995)
14. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. Materialized Views (1999)
15. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. Journal of Functional Programming (2006)
16. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming (2002)
17. Lee, K.Y., Son, J.H., Kim, M.H.: Efficient incremental view maintenance in data warehouses. In: CIKM (2001)
18. Liu, Y.A., Stoller, S.D., Gorbovitski, M., Rothamel, T., Liu, Y.E.: Incrementalization across object abstraction. In: OOPSLA (2005)
19. Maier, I., Odersky, M.: Deprecating the observer pattern in Scala.React. Technical report, EPFL Lausanne (2012)
20. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, Springer, Heidelberg (1991)
21. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: A programming language for ajax applications. In: OOPSLA (2009)
22. Morrison, D.R.: PATRICIA-Practical algorithm to retrieve information coded in alphanumeric. J. ACM (1968)
23. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Haskell (2002)
24. Prokopec, A., Bagwell, P., Rompf, T., Odersky, M.: A generic parallel collection framework. In: ICPP (2011)
25. Steele Jr., G.L.: Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In: ICFP (2009)
26. Sutherland, I.E.: Sketchpad: A man-machine graphical communication system. In: DAC (1964)
27. Willis, D., Pearce, D.J., Noble, J.: Caching and incrementalisation in the Java query language. ACM Sigplan Notices (2008)
28. Zanden, B.T.V., Halterman, R., Myers, B.A., Mcdaniel, R., Miller, R., Szekely, P., Giuse, D.A., Kosbie, D.: Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits (2001)

A Handbook for [ECOOP] PC Chairs

Giuseppe Castagna

Abstract. These notes describe how I organized the selection process for ECOOP. In particular they contain a list of tasks that are responsibility of the Program Committee (PC) chair before, during, and after the selection, as well as a description of the six-phases organization I used for selection.

Background. In these notes I summarize my experience about chairing ECOOP. Initially, I started them as a simple personal memorandum and vademecum to help me during the chairing task. As they grew in size and were enriched by external feedback, they became a sort of handbook that I intended to pass to my successors in order to spare them some head-aches (this is why the prose addresses to a future PC chair). Finally, several persons spurred me to make them available to a wider audience, which is why you find them in the "front matter" of the proceeding of ECOOP 2013. Whether you are a future (not necessarily ECOOP's) Program Committee Chair in search for new ideas and suggestions, or just a reader curious to know the behind-the-scenes of a program committee, I hope you will enjoy this reading.

Overview. For the future or soon-to-be PC chairs I want to stress that these notes are not intended to impose any particular organization of the PC but just as an help, to speed up the process and remind of a few points that may have been forgotten or overlooked. In particular you will find a lot of information that is missing from the *ECOOP PC chair FAQ* in the AiTO Handbook [1]. I would like to suggest you considering the way I handled PC submissions (Section 4) and organized the selection (Section 3), since I received positive returns on both of them. Of course, you have complete freedom to organize your PC as you like. Just, enrich these notes with your experience (I will be happy to provide the sources of this handbook) and pass them to the next PC chair.

The content of these notes is organized in four sections:

1. How to fix the dates
2. How to form a PC
3. How to organize the phases of the selection process
4. How to handle submissions of PC members

1 How to fix the dates

The first task of the PC Chair is to fix the following dates: (a.) submission deadline; (b.) rebuttal phase; (c.) PC meeting; (d.) notification; (e.) camera ready version.

a. Submission deadline: Although long-standing conferences have their deadlines in well-established time spans, you must nevertheless check that there is no overlap with the selection phases of related conference.

For ECOOP the submission deadline is traditionally in December.

> Criteria: if possible (*i*) *at least* one week (but 10 days are better) after ETAPS notifications (*ii*) not right before Christmas holidays since external reviewers are then hard to find (rule of thumb: no later than December the 15th.)

I strongly suggest to make this deadline (as well as the other dates) **strict**. Two expedients should help to maintain strictness: (1) explicitly state the strictness of the deadline in the call for paper and on the submission site and (2) put enough time between your submission deadline and the notification dates of related conferences (*eg*, for ECOOP, the notification dates of ESOP and FoSSaCS) so to have a safety cushion in case that any of these conferences does not respect its notification date.

b. Rebuttal: At least 7 weeks after the submission deadline. Reports must be available at least 9 days before the beginning of the rebuttal (safety for late arrivals, important time for a preliminary discussion on conflicting reports and for asking further external reviews to unravel these conflicts and/or to ensure at least one expert review for each paper: see the *Pre-rebuttal roundup phase* in Section 3); five days for bidding, two for assignment, one week for finding external reviewers and 4 weeks for preparing reviews. For ECOOP, however, consider that there will be Christmas holidays in the middle, so 8 weeks would be safer. Also avoid deadlines that are a couple of days before Christmas unless you are ready to accept that few PC members will complete their bids.

Set the rebuttal so that it includes at least one week-end day and one work day. Take into account time zones: make rebuttal start in the morning of Auckland and end in the night of San Francisco so that everybody has about the same amount of daylight to work on the rebuttal.

> Rule of thumb: 7 weeks after submission deadline excluding holidays.

c. PC meeting: If you organize a physical PC meeting —as customary for ECOOP—, then block a complete week at least twenty days after the end of the rebuttal. After you formed your Program Committee consult it to choose a couple of days in that week. You may use less than twenty days if you decide not to implement the *Middle round phase* of the reviewing process described in Section 3.

d. Notification: At least 3 days after the PC meeting, so that PC members have the time to travel back and update their reviews (though you'd better ask them to update their reviews during the meeting).

If you decide not to have a physical PC meeting and to carry on selection on line, then notification must be set at least four weeks after rebuttals

Remark: it is important to meet the notification deadline. Other conferences may have set their submission deadlines in function of your notification date.

In particular, for ECOOP, check that your notification deadline leaves a couple of weeks of margin to set the OOPSLA deadline (usually mid-April).

e. Final version: Springer requires camera ready to be sent 9 weeks before the start of the conference. So set the deadline for final versions *at least* 10 weeks before it. If your conference uses a different publisher, then check with it.

2 How to form a PC

Forming the PC is a delicate matter and it is advised to consult previous PC chairs for suggestions and advice, especially on the reliability of persons. In forming your PC you have to take into account different factors listed below.

a. Reliability of PC members. If you are a PC chair you have probably participated in a number of PC meetings already, so you know a good deal of people who are (or are not) reliable. I can never stress enough that in choosing a PC member, reliability is far more important than visibility. Ask previous PC chairs about particular persons they would recommend to include or exclude from the PC. Ask PC chairs of other conferences or persons you know for further confirmations if needed.

b. Spectrum. Make a list of areas your PC should cover and make sure you have two or three experts for each area. This list is very important since you will use it to assign papers (those that did not receive enough bids) and to form thematic discussion groups (see the *Middle round* phase in Section 3). Rule of thumb: enrich/modify the list of the previous edition (a lot of knowledge was already poured in it).

The geographical distribution of the PC members should be related to the geographical distribution of the authors who typically publish in your conference. Consider however that in case of a physical PC meeting (as it is tradition for ECOOP) members will be required to travel to attend the PC meeting, thus limit the number of PC members who have to travel more than 6 hours to attend it. Try to balance nationalities and consider gender representativity (as long as these aspects do not hinder scientific criteria). Try to add some "fresh blood": *at least* two or three young and brilliant researchers will be a real bonus, especially in a physical PC meeting, and so will few experienced researchers who were never invited to the PC before.

c. Statistics. Use previous editions' statistics about the spread of topics in submitted and accepted papers. Of course, put in your PC more experts for

topics that attract more submissions. And prepare statistics for the next PC chair.

A great tool to help you to prepare you PC is Frank Tip's PC-miner tool [2] which allows you to query a database of past publications and PC members of several conferences, among which is ECOOP. So for instance you can select all people that published at least 5 papers in ECOOP or OOPSLA but haven't served on the ECOOP committee yet.

d. Planning. When you send the invitations to take part in the PC you should include a clear plan and a list of commitments for the work ahead:

- Are submissions by PC members allowed? If so, outline how their selection will be organized and which criteria will be used (see Section 4).
- Ask for a strong commitment to be present at the PC meeting. The presence of all PC members at the meeting is **really** important so do not hesitate to insist on it.
- Expected work load:
  - Include a description of the phases of reviewing process and ask for commitments about availability for discussions.
  - Make it clear that there will be a rebuttal phase and that it is very important that external reviewers will be aware of it.
  - Describe the conflicts of interest
  - State your position about the use of external reviewers

For what concerns external reviewers I am quite in favor of using external reviewers. Actually, I suggest to ask every PC member to add to its personal review the review of an external reviewer whenever (s)he cannot produce an "X" (for eXpert) rated review (see the beginning of Section 3 for the classification of the reviews). While in principle this should ensure that all submissions will have an expert review, in practice this will not happen; nevertheless, this recommendation will reduce the number of reviews you will have to ask during the *Pre-rebuttal* phase (see Section 3).

As a side note, I suggest to send to PC members as few emails as possible and to repeat all important information in every mail: **never** assume that if you wrote something in a mail, then every member of the PC knows it (my personal experience was that many of the important pieces of information I wrote in my mails were missed by one or two members, not always the same ones).

f. Special PC members. Ask the steering committee to let you know the name of the next PC chair as soon as possible (before the paper submission deadline you can add to and remove from your PC as many members as you need) and invite her/him to join your Program Committee: it will be a very useful experience for her/him.

For ECOOP, it is a tradition that you invite to your ECOOP 20xx PC the PC chair of OOPSLA 20xx-1 (minus one, so if you chair ECOOP 2023, invite the chair of OOPSLA 2022... if both conferences still exist).

3 How to organize the phases of the selection process

Traditionally ECOOP follows the *Identify the Champion* policy [3]. In a nutshell, reviewers classify each submission from A to D with the meaning A: I will champion this paper at the PC meeting (advocate/accept); B: I can accept this paper, but I will not champion it (accept, but could reject); C: this paper should be rejected, though I will not fight strongly against it (reject, but could accept); D: serious problems, I will argue to reject this paper (detractor). Reviewers also classify their own expertise from X to Z (X: I am an expert; Y: I am knowledgeable in the area, though not an expert; Z: I am not an expert, my evaluation is that of an informed outsider). Then at the PC meeting only papers that are championed (at least one A) and without detractors (no D) are accepted: the discussion at the PC meeting is there to verify whether a submission satisfies these conditions. Oscar Nierstrasz explains in [3] how to prepare and organize the discussion for the PC meeting to implement such a policy.

Inasmuch as crucial the PC meeting is, it is only one of the several phases in which the selection process is organized. Your main role as a PC chair is to organize the reviewing process so that the decisions are taken on rational grounds. You can organize it as you think it is better, but here you are the methodology I followed for ECOOP 2013.

I organized the selection process in six phases: 1. Reviewing, 2. Pre-rebuttal roundup, 3. Rebuttal, 4. Middle round, 5. PC meeting, 6. Post meeting.

Phase 1: Reviewing

Ask PC members to return reviews as soon as possible and, in any case, *no later than 9 days before the start of the rebuttal phase.* Then

- As soon as the reports are in you should read them and try to clarify issues.
- You should spot reports that attribute a note without justification and ask the PC member responsible of the report to elaborate. Likewise, ask politely PC members who confuse review reports with Twitter tweets (and their 140 chars limit) to expand their reviews and give more details: explain that these are not just their personal notes for the PC meeting but must allow the PC chair and other persons of the committee to form an idea about the paper without reading it.
- You should check, as far as possible, that PC members are consistent in their evaluations.

The reviewing phase needs some preliminary organization, in particular you must prepare review forms and instructions so as it is clear that:

- there is a rebuttal phase and thus (1) deadlines are strict and (2) reviewers may ask direct questions to authors;
- PC members that cannot prepare an eXpert review should ask a further review to external *expert* reviewers;
- there will be specific periods in which external reviewers may be contacted again for further information.

Phase 2: Pre-rebuttal roundup

Pre-rebuttal roundup is the phase that takes place between the deadline for returning reviews and the start of the rebuttal phase. It is a crucial phase that will greatly influence the quality of the reviews you will work with and, thus, of your final program. I suggest to reserve 9 days for this phase, which must be used to:

1. ensure that every paper has at least 3 reports and at least an expert review and assign additional external reviewers if they do not.
2. smooth away divergences before rebuttals: ask reviewers with diverging opinions to react to each other reports (anonymized, for external ones). Assign external reviewers for further opinions.
3. [optional] start rejecting papers (personally, I prefer to do it in the *Middle round* phase, since I want to read rebuttals before taking any decision).

Last but not least this phase is an important safety cushion against late reviewers.

> Rationale for a roundup of 9 days. In my opinion 9 days are the perfect delay: if you have to ask an extra review you can ask the external reviewer to prepare it in a week so that it will be available for the rebuttal or, if that is not possible, you can ask it in 3 or 4 weeks so that it will be ready for the PC meeting.

The first two days of this phase are crucial: you must try spot most of the papers that need further reviews and send the corresponding requests to external reviewers in these two days.

It is *very important* that in this phase you carefully supervise the choice of external reviewers and monitor their work: ask for suggestions of external reviewers to all the PC members allocated to the paper and select the reviewer with them by a (very short) online discussion. In this way you will have the highest chances to get high quality external reviewers for all papers that need them.

Phase 3: Rebuttal

This phase will be a welcome pause in your activity.

- use the time of this phase to prepare the thematic groups for the next *Middle round* phase (see below).
- although several PC Chairs put a length limit on rebuttals because they are afraid to be obliged to "read a second paper", three past experiences where a limit was not imposed (ESOP 2011, POST 2012, and ECOOP 2013) suggest that it may not be useful to limit the length of the rebuttal, it suffices to ask (I suggest, firmly) authors to be concise and remind that the attention span of tired PC members will probably be not much longer than a few hundreds words.
- once the rebuttals are entered, try to spot not only the questions that were answered but, above all, the questions that were not answered.

Phase 4: Middle round

Exclude all papers that are very weak or very strong (and any PC member paper), and organize the remaining papers in *thematic groups* (alternatively you may decide to include in the thematic groups strong papers, too: see below).

Assign PC members to each group (asking for volunteers is neither fair nor appropriate and, probably, useless) provide some preliminary comparison in few synthetic points of the papers in the group and ask the PC members to assess the papers in their group. Assessing the papers of the group means:

> Browse (skim, have a quick look at) the papers, read the reviews, the rebuttals and the online discussions. If the reviewer feels competent on some paper of the group, the are *very* welcome to write an additional review. The aim is that for the final decision reviewers shall be able to give a (rough) relative ranking to the papers in their group.

The goal of thematic groups is that the best papers of each group should be accepted so as to ensure a good balance of topics and that all the spectrum of the conference is covered.

A secondary goal is to avoid to have at the PC meeting the classic 30mins-I'll-give-a-look-at-it reviews (I always feel uneasy with them). Other advantages are that with thematic groups PC members have a broader vision of the papers that will be discussed. And there is a core of few fixed persons that have a view of *all* undecided papers in a particular thematic area. In some sense you want to distribute over the whole PC the global vision that otherwise only the PC chair can have.

In my case I organized the papers in 7 thematic groups formed of 6 to 9 papers and assigned 4 PC members to each group. PC members were assigned to groups so as to maximize the number of papers they already reviewed in the group. The rationale is that this should have a strong correlation with their bids and experience (the more the papers they have in a group the more likely that they bidded for those papers and that they are experts in the topic). In any case do not expect to achieve a fair distribution.

> Rationale: whether to include strong papers in the thematic groups is a matter of trade-off. If you include them, then each group member will surely have a better vision of the theme, but in this way you also increase their charge of work and thus decrease the time they can spend on undecided papers. I preferred not to include strong papers in the thematic groups in the middle round phase.

Use also this phase to briefly discuss every weak paper (only C and D) and check whether everybody agrees not to discuss it in the PC meeting.

Finally, this phase is also used to decide submissions of PC members (see Section 4).

Phase 5: PC meeting

Exclude papers that have only C's and D's (you should have already agreed on that in the previous phase) and discuss all the others.

Plan 10 minutes of discussion for each paper and add one hour of buffer (if everything goes as planned, then PC members will use this time to update their reviews before leaving, otherwise you will be happy to have planned this extra hour). I strongly advice to ask somebody external from the PC to help you to keep track of the results of the discussion (you won't be able to do it during the meeting).

For the order in which papers are examined *Identify the Champion* [3] suggests to proceed from the best to the worse.[1] Frank Tip suggests to adopt a random order, instead. He gave me the following reasons:

> *I have done this at ISSTA'11 and PLDI'12 and I found it is better, because PC members won't automatically expect that the next paper will be accepted/rejected just because the previously discussed one was.*

I actually followed a mixed order. I grouped the papers to be discussed according to the thematic groups I used in the middle round (to which I added the strong papers) and then discussed the papers in each group ordered by their scores[2].

In my opinion this organization of the discussion in thematic groups has several advantages and a big drawback. The first advantage is that you will discuss papers on similar topic and with a consistent group of PC members. Second, PC members can profit that the discussion is on groups far from their expertise to take a little pause (typically to update their reviews or just to catch their breath: do not underestimate tiredness). Third, the rotation of thematic groups avoids the problem singled out by Frank Tip in the remark above. The big drawback is that according to the results of the first groups you will see that PC members will start panicking about the fact that the committee is accepting too many or too few papers. Therefore, this discussion order puts on you the burden of controlling all along the discussion that the final acceptance rate will be compatible with the format of the conference.

For the order of discussion of the groups you will not have much choice: it will depend on the PC members who are absent, leave earlier, arrive later, and if and when they can connect via video-conference. Rule of thumb: two or three PC members absent that leave in a close time zone, you can handle it (by video-conference); two PC members who live in a time zone 9 hours far from the place of the PC meeting, it gonna be tough; three of them absent and you can forget any reasonable organization.

In any case the order must be established at the beginning of the meeting so as people has time to prepare themselves for the discussion of the next paper.

[1] A paper is in the class XY if its higher score is X and lower score is Y and classes use the lexicographic order.

[2] I used the lexicographic order of (the alphabetically ordered juxtaposition of) their scores.

I distributed summaries for the whole discussion order and customized for each PC member (no excuse if you catch them distracted).

Finally, you will probably notice that your committee will use a slow the pace for the discussion at the beginning of the PC and tend to accelerate it at the end (because of tiredness). You must make sure that this will not happen so as to ensure a fair discussion to all papers.

A final word on organization. Logistics must be flawless since you cannot afford any delay in the progress of the meeting: choose a large room[3] with good wifi network and two (possibly, ceiling-mounted) video-projectors with their screens; test everything at least one week before the meeting (wifi, video, acoustic, videoconferencing) and send PC members their wifi passwords in advance; install enough power plugs for all PC members (rule of thumb: one and a half power plug for each PC member) and, if possible, have some plug converters ready for those who forgot it at home or at their hotel; have a back up for every important piece of hardware (laptop, wifi access point, omnidirectional microphones, web-cams, video-projectors) and configure them so that they are ready for use; make available at the meeting printed copies of all papers and of all important information (*eg*, discussion order, wifi passwords); do not neglect catering. Try to anticipate as many problems as possible, though problems will arise all the same.[4]

Phase 6: Post meeting

Once decisions are made, give 3 days to PC members to update their reviews so as they reflect the discussions in the PC meeting. However, Sophia Drossopoulou (chair of ECOOP '10) strongly suggests to ask PC members to update their reviews immediately after the meeting when memory is still fresh (the 3 days are just for safety) while James Noble (chair of ECOOP '12) strongly suggests to have it updated **during the meeting** (I did as suggested by James and the thematic group discussion order helped a lot in doing that).

You can send acceptance/rejection notifications right after the meeting and send the revised review reports a few days later.

Also if, as for ECOOP, you have/want to assign a best paper award, I suggest to do it by e-mail few days after the PC meeting using an *Election by Majority Judgment* [4]. In practice, make a list formed by half of the accepted papers that received the best scores (PC member papers excluded) and send it to PC members together with the following instructions:

1) Consult the joint list of XX papers (these are the accepted papers that received the best score from which I removed PC papers).

[3] To sit in a round 30 PC members (actually, in a U-shaped configuration with the video screens at the open side of the U) you will need a room for 60 persons (possibly not oblong).

[4] I spent two days testing everything and nevertheless the day of the meeting I discovered that one of the two wall screens of the room had been removed overnight: fortunately the painting on the wall was clear enough to project on it.

2) Pick as many papers in the list as you want. These papers must be chosen either because you think that they merit the award or because you think that they do not merit it.
3) Assign to each paper you chose a score from -2 (worse) to +2 (best) with the following meaning:
+2 It definitively deserves the award,
+1 A nice paper that may be awarded,
0 I'm neutral on this paper / I do not know,
-1 I am mildly against awarding this paper,
-2 It must not be awarded.
[not picking a paper is equivalent to give a score 0 to it]
4) Return your list of papers with their scores within a week

Then you sum all the scores of each paper and pick the paper with the highest score. Do not forget to notify the decision to all the persons concerned (above all the sponsors of the award: for ECOOP it is Springer).

4 How to handle submissions of PC members

Personally I am against PC member submissions because it does not seem fair to be judge and judged at the same time (whatever safety policy you impose) and I did not and will not submit to a conference in whose PC I take part in. However, there is a tradition in ECOOP to allow them. Actually, this is explicitly stated in the AiTO FAQ for ECOOP [1] that I quote:

> *Q: Are PC members allowed to submit papers?*
>
> *A: Yes, though the PC chair should not. PC papers are handled specially. The authors should not learn who reviewed their papers, and should leave the room when their papers are discussed at the PC meeting. Normally PC papers should be accepted only if they are of "above average" quality. In practice, this means that PC papers are rejected automatically if there is any objection from one of the referees.*

Furthermore there is a current trend in top-notch conferences to allow PC members to submit and use an External Reviewer Committees to handle them (*eg*, PLDI, POPL, OOPSLA). A couple of arguments also argue in favor of PC submissions:

1. Forbidding them would exclude a significant number of well-qualified leaders of the community from submitting (in ECOOP 2013 the submission that had the highest score —five A's— was coauthored by a PC member).
2. Persons that have a number of students who have work that they will be likely to submit to your conference may decline the invitation to take part in the PC. This would deprive the PC of the expertise of some leaders of the community.

Now, how to organize the review of PC member submissions? If you established an External Reviewers Committee (ERC), then this seems an obvious choice. For the rationale behind the ERC see Steve Blackburn's post on the subject [5]. I strongly prefer not to use an ERC since I do not want to limit a priori my choice for external reviewers.[5]

In ECOOP 2013 I established a different policy according to the following principles:

1. The policy to be followed for PC papers must be clearly and unambiguously established before submission time. I would say it will be the first thing that the PC will discuss. By discuss I mean, you have to propose a policy and *may* ask for suggestions of possible modifications to be sent directly to you: you then will synthesize them. Bottom line: avoid an open global discussion on such a delicate topic.
2. For what concerns the acceptance of a PC paper, after discussing it there must not be the slightest doubt that the PC paper deserves ECOOP publication.
3. Acceptance/rejection must be decided **before** the PC meeting (by e-mail, telephone, VOIP, ...). The reason for this (which I consider to be a key ingredient of the selection of PC papers) is threefold:
   (a) the quality of a PC paper must be established in absolute terms (we must be completely sure that if we take a PC paper it will not take the slot of a better paper discussed later in the PC meeting: this is the reason why they must be excluded from thematic groups, *cf.* phase 3 in Section 3),
   (b) I want to avoid people entering/exiting the PC meeting room, and to have a discussion about a paper of a person that 30 seconds before was sitting at the same table and is now waiting outside the room,
   (c) PC members with a submitted paper can discuss other papers freely without ulterior motive about how the discussion might influence the decision on their paper (unfortunately, that happens too, as Ken Birman confirmed me) since this decision is already taken.

   Important: try to schedule the time for discussing PC papers right after the assignment (or, in any case, as early as possible), because it may be quite difficult to find slots that fit everybody, especially if you want to hold a voice discussion. Also, try to have it in the first week after rebuttal and in any case as soon as possible after it: it is a good thing to leave some time between this decision and the PC meeting.

[5] Many people, whose opinion I highly regard, disagree with me on this point. Nevertheless I believe that if you stick to two points that I suggested in this handbook, namely, (1) ask PC members to find external expert reviewers when they cannot prepare an eXpert review themselves (*cf.* point *d* in Section 2) and (2) carefully supervise the choice of external reviewers in the pre-rebuttal phase by deciding their selection together with the concerned PC members (*cf.* phase 2 in Section 3), then you should obtain reviews of a quality higher than with an ERC.

4. Acceptance/rejection of PC papers will be communicated **after** the end of the PC meeting with all other notifications, so that a possible rejection of their paper cannot influence the behavior of PC members at the meeting (state this point quite clearly in your invitation email: *cf.* point *d* in Section 2).

Notice that in Section 1 I suggested to leave one month between the deadline for the reviews and the PC meeting, and three weeks between rebuttals and the PC meeting. This should give enough time to discuss PC papers.

As for what the exact meaning of "above average" or "higher criteria" for PC papers is, I think reasonable to request that they have a majority of A's (*after discussion*, so that these can be "upgraded" scores) and that any C must be compensated by a couple of very strong A's.

The important point here for me is that a PC paper must be *discussed* (of course only if it did not receive unanimously negative reviews), as well as be given the possibility to make a rebuttal. So, a PC paper that has a D can be accepted if for example the D turns out to be a low confidence reviewer that missed the point, or the rebuttal showed that the error spotted by the reviewer was not an error. On the contrary a PC paper that has only B's or that has only one lukewarm defender must not be accepted.

Acknowledgments. This handbook greatly benefited from suggestions from and discussions with Sophia Drossopoulou, Ken Birman, Richard Jones, Shriram Krishnamurthi, James Noble, and Frank Tip.

References

1. Nierstrasz, O., Kappel, G., Vasconcelos, V.: The AiTO Handbook, `http://www.aito.org/resources/handbook.pdf`
2. Tip, F.: PC miner, `https://cs.uwaterloo.ca/~ftip/pcminer.html`
3. Nierstrasz, O.: Identify the Champion. In: Pattern Languages of Program Design, vol. 4, pp. 539–556. Addison-Wesley (2000), `http://scg.unibe.ch/download/champion/`
4. Balinski, M., Laraki, R.: Election by Majority Judgement: Experimental Evidence. In: Situ and Laboratory Experiments on Electoral Law Reform. Studies in Public Choice, vol. 25, pp. 13–54. Springer (2011), `http://hal.archives-ouvertes.fr/docs/00/24/30/76/PDF/2007-12-18-1691.pdf`
5. Blackburn, S.: Why Have an External Review Committee? `http://users.cecs.anu.edu.au/~steveb/research/review-committee`

Author Index